How to Write a Wordpress Plugin
Written by Ronald Huereca *(http://www.ronalfy.com/)*

About this series:
How to Write a Wordpress Plugin, written by Ronald Huereca is an extensive, twelve part series on the process of creating your own Wordpress plugin. Every step is covered, from "*Seven Steps for Writing a Wordpress Plugin*" all the way down to adding ajax to your plugin and releasing it. This is an excellent article series for anyone interested in the process behind creating your very first Wordpress plugin. With code examples to help assist you, you will be on your way to future releases of your own plugins for the Wordpress community.

About the author:
Ronald is frequently found laying his thoughts out in strong, straight-forward articles on various web related topics. He comes from a relatively strong technical and business background, having an undergrad in Electrical Engineering Technology and a Master of Science in Business Administration. A programmer by day, and web hobbyist (and writer) by night, who also runs his own blog at www.ronalfy.com. Ronald has been on the Devlounge team since the fall of 2006, and has contributed many wonderful articles, including this very wordpress series. He also writes for Weblogtoolscollection.

Introduction

For any WordPress user, plugins are essential. WordPress Plugins allow those with little to no programming skills to extend the functionality of their blog. Plugins come in all shapes and sizes, and there is a plugin that does just about anything for WordPress.

As good as WordPress is as a standalone application, there are still things that WordPress lacks. Users are requesting more and more features for WordPress that would be very feasible to write as a plugin. There are many untapped ideas out there, and new ones created every day.

Having released three plugins already (not counting the custom ones I wrote), I am aware of some of the limitations of WordPress and wish to share some of the lessons I have learned (and am still learning) about creating WordPress plugins. As a result, I will be starting series that will discuss various topics regarding writing your own WordPress plugin. The series will start off very introductory and will assume your plugin knowledge is zilch.

Who is this Series For?

This series is for any WordPress user who is curious about or wants to learn how to write their own WordPress plugin. Readers of this series will have an intermediate knowledge of PHP, know a little JavaScript, and be decent at Cascading Style Sheets.

This plugin series will benefit theme designers, those that like to tinker with plugin code, and those that are interested in writing their own plugin from scratch.

Tools to Get the Job Done

To write plugins, any text editor will do. Here are the tools I personally use to create plugins.

- Dreamweaver
- Firefox
- Firebug Firefox Extension
- Web Developer Firefox Extension
- XAMPP with a local WordPress installation

This series assumes you have WordPress 2.1.x or greater installed.

Code Samples

All code I use will be available for download after each post in the **Conclusion** section. I will be building the code as I go along, so each download will be different. I will be creating a plugin that doesn't really do anything other than to show you the basics of how a plugin works.

Since each post in this series builds on top of each other, it is recommended to read this series in the order it is presented.

I highly recommend not using the test plugin on a production WordPress installation. Instead, use a local WordPress installation.

Topics

I plan to start off really basic and move quickly into the more hard-core WordPress plugin functions. This series will not be a comprehensive micro-detail of plugin development, but will hopefully give you a nice foundation to start your own plugin development. If you have any questions or suggestions, please leave a comment or e-mail me using the Devlounge contact form (Ronald). I do ask that you not rely on Devlounge for support and instead use the WordPress Support forums.

Techniques

Some of the techniques I use in my code samples may not be the best way to present code and you may be cringing because I don't have a lot of shortcuts. I apologize in advance. Everybody has a different coding style.

As far as plugin techniques, structure, behavior, and other nuisances, if there is a better and easier way that I overlooked, I am all ears (er, eyes).

Seven Reasons to Write a Wordpress Plugin

While writing the “How to Write a Plugin” series, I thought it would be beneficial to list some reasons why WordPress users would want to write a WordPress plugin in the first place.

Listed below are seven reasons why a WordPress user should consider writing a WordPress plugin.

You like a plugin’s idea, but don’t like the plugin’s implementation

Whether discovering WordPress plugins on Weblog Tools Collection, the official WordPress plugins directory, or the WordPress Plugin Database, you will inevitably find a plugin that meets your needs — sort of.

You like the idea of the plugin, but not really the approach the plugin author took with it. Why not run with the original idea and create your own separate version?

You want to modify existing plugin code

Sometimes the plugin’s output needs to be tweaked a little bit or some functionality you would like is missing. You can try convincing the plugin author to add your feature, but plugin authors are usually quite busy or they may not like your suggestion. It takes a lot of effort by a plugin author to provide support and field feature and bug requests for a plugin that is free. Sometimes the plugin is no longer supported by anyone.

In the event the plugin author is unable to your needs, it will be up to you to take the initiative and modify the existing plugin code. If you do a good enough job and make enough changes, you can re-release the plugin as long as the original plugin was released under a GPL compatible license.

Usually one of the first things I do when I install or test a new plugin is to look at the code and see what I can modify, what I can’t modify, and what I can possibly add or take away.

You want to extend a plugin

Sometimes a plugin is good as it is, but you would like to build upon it and release your own version. For example, you may think a plugin would work better using AJAX, or would like to add more hooks so that it is compatible with other plugins. You may want to add an admin panel so you don’t have to dig through the code to change the output.

As stated earlier, if a plugin is released as GPL compatible, you are free to release your own version.

You want portable theme code

For those of us who opted to build a custom theme from scratch rather than download one, you may find yourself re-using code snippets all over the place. Wouldn’t it be

better just to write your own plugin that combined all the little code snippets so that you could use them as template tags?

The beauty of template tags is that you can re-use them over and over for your theme and any future ones you build. And you only have one place to change the code rather than several.

You are a theme designer

I would argue that if you are a template designer for WordPress, the next logical step is to be a plugin author. Writing plugins gives you a more intimate knowledge of how WordPress behaves and allows you to extend the functionality of your released themes.

You want to make money

A good plugin author can usually get paid on the side for custom work. Some plugin authors take donations as well or charge extra for providing support or for consulting.

If you are a custom theme designer, you can package your custom plugins in with the theme for an extra charge.

You want incoming links

When launching the Reader Appreciation Project, one of the goals I had was to rapidly build incoming links. The best way I knew how was to write some WordPress plugins and promote them. One of my plugins (WP Ajax Edit Comments) turned out to be very popular and has currently generated more than 100 incoming links.

How to get ideas for Wordpress Plugins

If you are convinced that you would like to investigate the possibility of creating your own WordPress plugin, it may be hard to think of that idea that will allow you to take the plunge. Fortunately, there are many places to find inspiration regarding developing your own WordPress plugin. Within this post, I will list several ways to get ideas for your very own WordPress plugin.

Listen to your Readers

Your readers are a valuable asset when it comes to getting ideas for plugins. For example, a reader might request an easy way to reply to or edit comments. Since blog readers are the ones who use your blog the most, they have a unique insight in what they want out of your blog. Just the other day, one of my readers asked me to have a way to preview a comment before posting. Luckily there is already a few plugins out there for that, but sometimes your readers will suggest something that has yet to be implemented as a plugin.

Listen to Yourself

“If only WordPress could do…”

If you find that WordPress lacks a feature that you truly want, why not program it yourself in the form of a plugin? Chances are that if you desire the feature added, others will too.

Check out Blogging Resources

Sites such as The Blog Herald and Weblog Tools Collection are great resources for plugin ideas. On Wednesday, The Blog Herald has a column called WordPress Wednesday. Within this column are plugin requests and a “wishlist” for WordPress. Weblog Tools Collection typically has a plugin announcement almost every day, and from there you can get an idea of what kind of plugins people are churning out.

Check out the WordPress Support Forums

The WordPress Support Forums are full of people looking for help on extending their WordPress blog. A particularly useful forum for plugin ideas is the Requests and Feedback forum. Another area is the WordPress ideas page.

Investigate APIs

Online services such as Flickr, FeedBurner, Google Maps, and others have APIs (Application Program Interfaces) that allow third-party applications the ability to interface with their services. Through these APIs, you start programming your own WordPress solution.

If there is a service that you really like, but you would like to see it included in WordPress, investigate the service’s API and see if it would make a good plugin.

Third Party Applications

There are many third-party applications that people may have installed along with a WordPress blog. Examples of such programs are Mint, Vanilla, and many others. Why not develop a WordPress plugin that integrates these third-party applications into a WordPress blog?

Existing WordPress Plugins

If you find a WordPress plugin you really like and would like to branch out with your own idea, feel free to do so. If you don't like the implementation of a particular plugin, build your own implementation. There are many plugins out there that essentially do the same thing, but are all slightly different.

Structure of a Wordpress Plugin

One of the more important aspects of developing a WordPress plugin is how you structure it. This post will go over some tips on how to structure your plugin to organize your plugin resources and avoid naming collisions. Each plugin author is different in the way they structure a plugin, so these tips are merely my own personal preference. I'll first briefly describe how a WordPress plugin works and then go into a plugin's structure.

How a WordPress Plugin Works

After placing a WordPress plugin into the "wp-content/plugins/" folder, the plugin should automatically be available to install.

When a plugin is "Activated", this tells WordPress to load your bit of code on "each" page (including admin pages). This is why if you have many plugins activated, your WordPress installation may be very slow due to the amount of code being included.

Since WordPress loads your code automatically when the plugin is activated, you can take advantage of this by tapping into the WordPress Plugin Application Program Interface (API). You can also access the WordPress template tags or create your own.

I suggest reading into the WordPress loop if you plan on making changes to the post content or comments. The WordPress loop is the loop that displays your posts. Some template tags will not work outside of this loop, so it is imperative that you know exactly where your code is executing. You can control this by taking advantage of actions and filters, which will be explained in later posts.

Folder Structure

All WordPress plugins will be installed in the wp-content/plugins directory. Some plugin authors simply include a PHP file for their plugin, but I recommend always creating a folder to store your plugin.

I typically structure my plugin in this folder structure:

- Plugin Folder Name (The name of your plugin with no spaces or special characters)
    - Main plugin php file
    - js folder (for JavaScript files)
    - css folder (for StyleSheet files)
    - php folder (for other PHP includes)

For example purposes, here is a sample structure I have created:

- devlounge-plugin-series
    - devlounge-plugin-series.php
    - js
    - css
    - php

Within the **devlounge-plugin-series** folder, I would include just the main PHP file and put all other files in their respective folders. This structure will assist other plugin authors who look at your code to be able to tell what the main plugin file is and where all the supporting files are located.

WordPress also recommends placing images in their own directory and including a read me file for your plugin.

Main Plugin File

When you start a new plugin file, the first seven lines are the lines that describe your plugin.

PHP:

```
1. <?php
2. /*
3. Plugin Name: Your Plugin Name Here
4. Plugin URI: Your Plugin URI
5. Version: Current Plugin Version
6. Author: Who Are You?
7. Description: What does your plugin do?
```

Line 3 allows you to name your plugin. Line 4 allows you to point a user to the web location of your plugin. Line 5 allows you to specify the current version. Line 6 allows you to specify the author of the plugin. Line 7 allows you to describe your plugin.

Shown below is an example of the code filled out:

PHP:

```
1. <?php
2. /*
3. Plugin Name: Devlounge Plugin Series
4. Plugin URI: http://www.devlounge.net/
5. Version: v1.00
6. Author: <a href="http://www.ronalfy.com/">Ronald Huereca</a>
7. Description: A sample plugin for a <a
   href="http://www.devlounge.net">Devlounge</a> series.
```

Shown below is a screenshot of what the plugin would look like in the WordPress Plugins panel.

Set Up a Class Structure

You don't have to be incredibly familiar with PHP Classes to develop a WordPress plugin, but it sure helps. A class structure is necessary in order to avoid naming collisions with other WordPress plugins. If someone out there sets up the same function name as yours in a plugin, an error will result and WordPress will be rendered inoperable until that plugin is removed.

To avoid naming collisions, it is imperative that all plugins incorporate a PHP class structure. Here is some bare-bones code that will allow you to set up a class structure.

PHP:

```
1. if (!class_exists("DevloungePluginSeries")) {
2.     class DevloungePluginSeries {
3.         function DevloungePluginSeries() { //constructor
4.
5.         }
6.
7.     }
8.
9. } //End Class DevloungePluginSeries
```

What the above code does is checks for the existence of a class named **DevloungePluginSeries**. If the class doesn't exist, the class is created.

Initialize Your Class

The next bit of code will initialize (instantiate) your class.

PHP:

```
1. if (class_exists("DevloungePluginSeries")) {
2.     $dl_pluginSeries = new DevloungePluginSeries();
3. }
```

All the above code checks for is if the class **DevloungePluginSeries** has been created. If it has, a variable called **$dl_pluginSeries** is created with an instance of the **DevloungePluginSeries** class.

Set Up Actions and Filters

The next bit of code sets up a place holder for WordPress actions and filters (which I will go over in a later post).

PHP:

```
1. //Actions and Filters
2. if (isset($dl_pluginSeries)) {
3.     //Actions
4.
```

```
    //Filters
}

?>
```

The above code checks to make sure the **$dl_pluginSeries** variable is set. If it is (and that's only if the class exists), then the appropriate actions and filters are set up.

Wordpress Plugin Actions

WordPress actions allow you as a plugin author to be able to hook into the WordPress application and execute a piece of code. An example of an action would be that you want a execute some code after a user has published a post or left a comment.

Some of the actions that I use heavily are:

- admin_menu: Allows you to set up an admin panel for your plugin.
- wp_head: Allows you to insert code into the <head> tag of a blog

Actions in Action

While defining the structure of a WordPress plugin, I left a place holder for some actions. In this example, we are going to set up a piece of code that will run inside the <head> tag of a WordPress blog.

First we need to add a function into our **DevloungePluginSeries** class.

PHP:

```
1.  function addHeaderCode() {
2.          ?>
3.  <!-- Devlounge Was Here -->
4.          <?php
5.
6.      }
```

All the above function does is output an HTML comment. Rather simple, but you could output just about anything. To call this function, we add an action.

PHP:

```
1.  //Actions and Filters
2.  if (isset($dl_pluginSeries)) {
3.     //Actions
4.     add_action('wp_head', array(&$dl_pluginSeries, 'addHeaderCode'), 1);
5.     //Filters
6.  }
```

From the WordPress Plugin API page, the add_action structure is as follows:
add_action ('hook_name', 'your_function_name', [priority], [accepted_args]);

Since we are calling a function inside of a class, we pass the action an array with a reference to our class variable (**dl_pluginSeries**) and the function name we wish to call (**addHeaderCode**). We have given our plugin a priority level of 1, with lower numbers executed first.

Running the Code

If the Devlounge Plugin Series plugin is activated, the comment of "Devlounge was here" should show up when you go to View->Source in your web browser when looking at your main blog site.

Removing Actions

If your plugin dynamically adds actions, you can dynamically remove actions as well with the remove_actions function. The structure is as follows:
remove_action('action_hook','action_function').

Wordpress Plugin Filters

WordPress filters are the functions that your plugin can hook into with regards to modifying text. This modified text is usually formatted for either inserting into a database or displaying the output to the end user.

WordPress filters allow to you modify virtually any kind of text displayed and are extremely powerful. WordPress filters allow you to modify posts, feeds, how authors are displayed in comments, and much, much more.

To demonstrate the usefulness of WordPress filters, we will continue working with the existing code in the Devlounge Plugin Series code from the WordPress Plugin Actions post.

Adding A Content Filter

One of the cool filters you can hook into is one called '**the_content**'. This particular filter is run for post content being displayed to the browser. We're going to just add a line of text that will be displayed at the end of the content.

The format for adding a filter from the WordPress Plugin API is: add_filter('hook_name', 'your_filter', [priority], [accepted_args]);

We just need to add in a function to the **DevloungePluginSeries** class. Let's call it **addContent**.

PHP:

```
1.  function addContent($content = '') {
2.              $content .= "<p>Devlounge Was Here</p>";
3.              return $content;
4.          }
```

In the above code, the following things are happening.

- The above function will accept one variable named **content**.
- If no variable is passed, a default value is set.
- The **content** variable has our line of text added to it.
- The text is then returned.

After the function is added to the class, the next step is to hook into the '**the_content**' filter and call the function.

PHP:

```
1.  //Actions and Filters
2.  if (isset($dl_pluginSeries)) {
3.      //Actions
4.      add_action('wp_head', array(&$dl_pluginSeries, 'addHeaderCode'), 1);
```

```
5.      //Filters
6.      add_filter('the_content', array(&$dl_pluginSeries, 'addContent'));
7.  }
```

As you can see on line 6, a filter with the name '**the_content**' is added and our function '**addContent**' is called.

If the plugin were activated and a post was viewed, the text "Devlounge Was Here" would show up towards the end of the post content.

Adding An Author Filter

Another example of a filter I will show is manipulating the display of comment authors. I'm simply going to make all authors uppercase.

We just need to add in a function to the **DevloungePluginSeries** class. Let's call it **authorUpperCase**.

PHP:

```
1.  function authorUpperCase($author = '') {
2.              return strtoupper($author);
3.          }
```

In the above code, the following things are happening.

- The above function will accept one variable named **author**.
- If no variable is passed, a default value is set.
- The **author** string is returned as uppercase.

After the function is added to the class, the next step is to hook into the '**get_comment_author**' filter and call the function.

PHP:

```
1.  //Actions and Filters
2.  if (isset($dl_pluginSeries)) {
3.      //Actions
4.      add_action('wp_head', array(&$dl_pluginSeries, 'addHeaderCode'), 1);
5.      //Filters
6.      add_filter('the_content', array(&$dl_pluginSeries, 'addContent'));
7.      add_filter('get_comment_author', array(&$dl_pluginSeries,
    'authorUpperCase'));
8.  }
```

As you can see on line 7, a filter with the name '**get_comment_author**' is added and our function '**authorUpperCase**' is called.

If the plugin were activated and a post with comments was viewed, the comment authors would all be upper case.

Applying Filters

One of the more powerful things you can do with filters is to call then dynamically. There's no need to add a filter to be run every time. You can run a filter whenever you choose from within your code.

The format for the **apply_filters** function is: apply_filter('filter name', 'your text');

You will see an example of **apply_filters** in use later in this series.

Constructing a Wordpress Plugin Admin Panel

Any plugin that needs user input, such as changing a variable, should have some kind of administration panel. Building a administration panel isn't all that difficult, so it annoys me when plugin authors decide not to build one and want plugin users to modify PHP code. Asking users -- whose experience with PHP might be nil -- to modify code is generally not a good idea. This post will go into what it takes to successfully create an admin panel for your plugin.

A Place to Store the Variables

One of the first problems you will likely encounter when constructing your own admin panel is where exactly to store the variables. Luckily WordPress makes it quite easy with options. I will explain options and database storage in a later post in this series. For now, all you have to do is nod your head and follow the steps to store your own admin variables in the WordPress database.

The first thing I usually do with regards to options is to assign a "unique" name for my admin options. I store this in the form of a member variable inside my class. In the case of the Devlounge Plugin Series plugin, I added this variable declaration to the **DevloungePluginSeries** class:

Name Your Admin Options

PHP:

```
1.  class DevloungePluginSeries {
2.      var $adminOptionsName = "DevloungePluginSeriesAdminOptions";
3.      function DevloungePluginSeries() { //constructor
4.
5.      }
```

Line 2 shows where I added in my member variable. I named my variable **adminOptionsName** and gave it the long and unique value of **DevloungePluginSeriesAdminOptions**.

Set Your Admin Default Options

You're going to need a place to initialize your admin options, especially when a user first activates your plugin. However, you also need to make these options upgrade-proof just in case you decide to add more options in the future. My technique is to provide a dedicated function to call your admin options. Your plugin needs may be different, but most admin panels aren't incredibly complicated so one function for your admin options should be sufficient.

Here's the function I inserted in the **DevloungePluginSeries** class:

PHP:

```
1.  //Returns an array of admin options
2.      function getAdminOptions() {
3.          $devloungeAdminOptions = array('show_header' => 'true',
4.              'add_content' => 'true',
5.              'comment_author' => 'true',
6.              'content' => '');
7.          $devOptions = get_option($this->adminOptionsName);
8.          if (!empty($devOptions)) {
9.              foreach ($devOptions as $key => $option)
10.                 $devloungeAdminOptions[$key] = $option;
11.         }
12.         update_option($this->adminOptionsName, $devloungeAdminOptions);
13.         return $devloungeAdminOptions;
14.     }
```

What this function does is:

- Assigns defaults for your admin options (lines 3 - 6).
- Attempts to find previous options that *may have* been stored in the database (line 7).
- If options have been previously stored, it overwrites the default values (lines 8 - 11).
- The options are stored in the WordPress database (line 12).
- The options are returned for your use (line 13).

Initialize the Admin Options

The **getAdminOptions** can be called at anytime to retrieve the admin options. However, what about when the plugin is first installed (er, activated)? There should be some kind of function that is called that also retrieves the admin options. I added the following function into the **DevloungePluginSeries** class:

PHP:

```
1. function init() {
2.         $this->getAdminOptions();
3.     }
```

Short, sweet, and simple. An action, however, is required to call this **init** function.

PHP:

```
1. add_action('activate_devlounge-plugin-series/devlounge-plugin-series.php',
   array(&$dl_pluginSeries, 'init'));
```

This action is a little complicated, but easy to figure out. Here's what the action does:

- You tell it to run when a plugin has been activated.
- You give it the path to the main plugin PHP file, which is **devlounge-plugin-series/devlounge-plugin-series.php**. This of course is assuming that your plugin is properly placed in the **wp-content/plugins/** directory.
- You pass a reference to the instance variable **dl_pluginSeries** and call the **init** function.

So anytime the plugin is activated, the **init** function is called for the Devlounge Plugin Series plugin.

How the Admin Panel Works

Before I delve into the code of constructing the actual admin panel, it will be beneficial to describe the behavior of the admin panel. Here are the steps you'll want to take for setting up your admin panel:

- Check to see if any form data has been submitted.
- Output notifications if form data is present.
- Display the admin panel options.

One thing that may confuse you greatly in the admin panel is the use of the **_e** WordPress function. The **_e** function allows WordPress to search for a localized version of your text. This will help WordPress potentially translate your plugin in the future. The function works like a normal echo, but instead you pass it your text and an identifier variable (typically your plugin name). An example would be:

```
_e('Update Settings', 'DevloungePluginSeries')
```

This code would work the same way as: echo "Update Settings".

Set up the Admin Panel Function

The first thing we want to do is set up a function that will actually print out the admin panel. The function's name will be **printAdminPage**. This next bit of code will read in the options we specified earlier and check to see if any post options have been submitted. All the code in this section is assumed to be within the **printAdminPage** function.

PHP:

```
1.  //Prints out the admin page
2.      function printAdminPage() {
3.              $devOptions = $this->getAdminOptions();
4.
5.              if (isset($_POST['update_devloungePluginSeriesSettings'])) {
6.                  if (isset($_POST['devloungeHeader'])) {
7.                      $devOptions['show_header'] = $_POST['devloungeHeader'];
8.                  }
9.                  if (isset($_POST['devloungeAddContent'])) {
10.                     $devOptions['add_content'] =
    $_POST['devloungeAddContent'];
11.                 }
12.                 if (isset($_POST['devloungeAuthor'])) {
13.                     $devOptions['comment_author'] = $_POST['devloungeAuthor'];
14.                 }
15.                 if (isset($_POST['devloungeContent'])) {
16.                     $devOptions['content'] = apply_filters('content_save_pre',
    $_POST['devloungeContent']);
17.                 }
18.                 update_option($this->adminOptionsName, $devOptions);
19.
20.                 ?>
21. <div class="updated"><p><strong><?php _e("Settings Updated.",
    "DevloungePluginSeries");?></strong></p></div>
22.             <?php
23.             } ?>
```

All the above code does is load in the options and test to make sure each portion of the form is submitted. The if-statement overkill isn't necessary, but sometimes it is useful for debugging. The first form variable that is tested as being set is **update_devloungePluginSeriesSettings**. This variable is assigned to our "Submit" button. If that isn't set, it's assumed that a form hasn't been submitted.

As promised, in line 16, I use the **apply_filters** function to format the content for saving in the database.

The next bit of code will display the HTML form that is necessary for the admin panel. It's a little involved, so I'll summarize it here. All the code is doing is displaying the form elements and reading in options.

PHP:

```
1.  <div class=wrap>
2.  <form method="post" action="<?php echo $_SERVER["REQUEST_URI"]; ?>">
3.  <h2>Devlounge Plugin Series</h2>
4.  <h3>Content to Add to the End of a Post</h3>
5.  <textarea name="devloungeContent" style="width: 80%; height: 100px;"><?php
    _e(apply_filters('format_to_edit',$devOptions['content']),
    'DevloungePluginSeries') ?></textarea>
6.  <h3>Allow Comment Code in the Header?</h3>
7.  <p>Selecting "No" will disable the comment code inserted in the header.</p>
8.  <p><label for="devloungeHeader_yes"><input type="radio"
    id="devloungeHeader_yes" name="devloungeHeader" value="true" <?php if
    ($devOptions['show_header'] == "true") { _e('checked="checked"',
    "DevloungePluginSeries"); }?> /> Yes</label>    <label
    for="devloungeHeader_no"><input type="radio" id="devloungeHeader_no"
    name="devloungeHeader" value="false" <?php if ($devOptions['show_header']
    == "false") { _e('checked="checked"', "DevloungePluginSeries"); }?>/>
    No</label></p>
9.
10. <h3>Allow Content Added to the End of a Post?</h3>
11. <p>Selecting "No" will disable the content from being added into the end of a
    post.</p>
12. <p><label for="devloungeAddContent_yes"><input type="radio"
    id="devloungeAddContent_yes" name="devloungeAddContent" value="true"
    <?php if ($devOptions['add_content'] == "true") { _e('checked="checked"',
    "DevloungePluginSeries"); }?> /> Yes</label>    <label
    for="devloungeAddContent_no"><input type="radio"
    id="devloungeAddContent_no" name="devloungeAddContent" value="false"
    <?php if ($devOptions['add_content'] == "false") { _e('checked="checked"',
    "DevloungePluginSeries"); }?>/> No</label></p>
13.
14. <h3>Allow Comment Authors to be Uppercase?</h3>
15. <p>Selecting "No" will leave the comment authors alone.</p>
16. <p><label for="devloungeAuthor_yes"><input type="radio"
    id="devloungeAuthor_yes" name="devloungeAuthor" value="true" <?php if
    ($devOptions['comment_author'] == "true") { _e('checked="checked"',
    "DevloungePluginSeries"); }?> /> Yes</label>    <label
    for="devloungeAuthor_no"><input type="radio" id="devloungeAuthor_no"
    name="devloungeAuthor" value="false" <?php if
    ($devOptions['comment_author'] == "false") { _e('checked="checked"',
    "DevloungePluginSeries"); }?>/> No</label></p>
17.
18. <div class="submit">
19. <input type="submit" name="update_devloungePluginSeriesSettings"
    value="<?php _e('Update Settings', 'DevloungePluginSeries') ?>" /></div>
20. </form>
21.  </div>
22.                 <?php
23.             }//End function printAdminPage()
```

One observation to make in the above code is the reference to the options and how the HTML and PHP is integrated.

Set up the Admin Panel Action

Now the the **printAdminPage** function is added, we need to call it through an action. First a function must be set up right above the actions that it outside the scope of the class.

PHP:

```
1.  //Initialize the admin panel
2.  if (!function_exists("DevloungePluginSeries_ap")) {
3.      function DevloungePluginSeries_ap() {
4.          global $dl_pluginSeries;
5.          if (!isset($dl_pluginSeries)) {
6.              return;
7.          }
8.          if (function_exists('add_options_page')) {
9.      add_options_page('Devlounge Plugin Series', 'Devlounge Plugin Series', 9,
    basename(__FILE__), array(&$dl_pluginSeries, 'printAdminPage'));
10.         }
11.     }
12. }
```

The above code does this:

- A function named **DevloungePluginSeries_ap** is created.
- Variable **dl_pluginSeries** is tested for existence (lines 4 - 7). This variable references our class.
- An admin page named "Devlounge Plugin Series" is initialized and our **printAdminPage** function is referenced (lines 8-10).

The **add_options_page** function format is described by WordPress as:
add_options_page(page_title, menu_title, access_level/capability, file, [function]);

The access level (in this case a 9) is described in more detail at the Users Levels page in the WordPress codex.

An action must now be set up to call the **DevloungePluginSeries_ap** function:

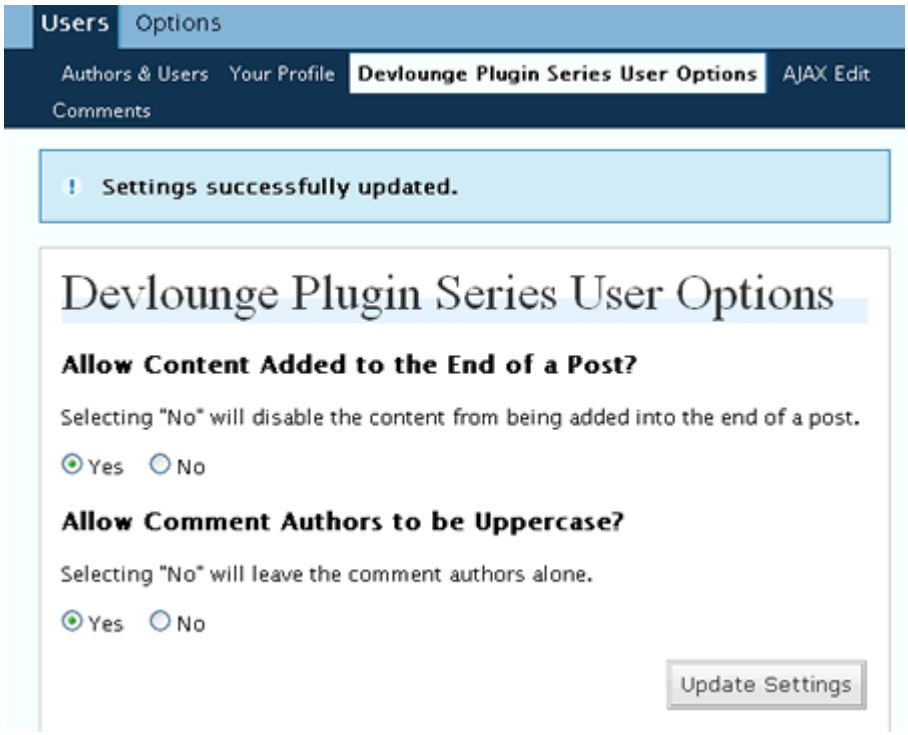
PHP:

```
1.  add_action('admin_menu', 'DevloungePluginSeries_ap');
```

Constructing a Wordpress Plugin User's Panel

There will be situations where you will have a main administrative panel, but would like individual users to set their own preferences. In the case of the Devlounge Plugin Series, we added an option for text to be added in at the end of each post. However, what if a logged-in user doesn't want to see this text? Why not give them the option without affecting all of the other users?

This post will go over the steps to add in your own User's Administration Panel.



Name Your User Options

PHP:

```
1.  class DevloungePluginSeries {
2.          var $adminOptionsName = "DevloungePluginSeriesAdminOptions";
3.          var $adminUsersName = "DevloungePluginSeriesAdminUsersOptions";
```

Line 3 shows where I added in the member variable called **adminUsersName** . I gave this variable the long and unique name of **DevloungePluginSeriesAdminUsersOptions**.

Set Your the Default User Options

You're going to need a place to initialize your user options, especially when a user first activates your plugin. However, these options should also work outside of the admin panel where users may or may not be logged in.

Here's the function I inserted in the **DevloungePluginSeries** class:

PHP:

```
1.  //Returns an array of user options
2.      function getUserOptions() {
3.          global $user_email;
4.          if (empty($user_email)) {
5.              get_currentuserinfo();
6.          }
7.          if (empty($user_email)) { return ''; }
8.          $devOptions = get_option($this->adminUsersName);
9.          if (!isset($devOptions)) {
10.             $devOptions = array();
11.         }
12.         if (empty($devOptions[$user_email])) {
13.             $devOptions[$user_email] = 'true,true';
14.             update_option($this->adminUsersName, $devOptions);
15.         }
16.         return $devOptions;
17.     }
```

What this function does is:

- Checks to see if a user is logged in (lines 3 - 7). This is easily determined by checking to see if the **user_email** variable is set.
- Attempts to find previous options that *may have* been stored in the database (line 8).
- If options aren't found, defaults are assigned (lines 9-15)
- The options are returned for your use (line 16).

Initialize the Admin User Options

The **getUserOptions** can be called at anytime to retrieve the admin user options. However, what about when the plugin is first installed (er, activated)? There should be some kind of function that is called that also retrieves the user options. I added the following function into the **init** function:

PHP:

```
1.  function init() {
2.          $this->getAdminOptions();
3.          $this->getUserOptions();
4.      }
```

Line 3 calls the new function **getUserOptions**. Since there is already an action added that calls the **init** function, no extra steps are necessary.

How the Admin Panel and User Panel Will Work Together

You will recall from the last post about setting up an admin panel that the WordPress admin could set the content at the end of the post, whether code was shown in the header, and whether an author's name was uppercase in the comments. The user's panel allows users who *aren't admin* to be able to specify whether they want these options or not.

We're going to allow the user to decide if they:

- Want content at the end of the post to show (only if the admin has this enabled already).
- Wants the comment authors to be uppercase (only if the admin has this enabled already).

Set up the User's Panel Function

The first thing we want to do is set up a function that will actually print out the user's panel. The function's name will be **printAdminUsersPage**. This next bit of code will read in the options we specified earlier and check to see if any post options have been submitted. All the code in this section is assumed to be within the **printAdminUsersPage** function.

PHP:

```
1.  //Prints out the admin page
2.              function printAdminUsersPage() {
3.                  global $user_email;
4.                  if (empty($user_email)) {
5.                      get_currentuserinfo();
6.                  }
7.                  $devOptions = $this->getUserOptions();
8.
9.                  //Save the updated options to the database
10.                 if (isset($_POST['update_devloungePluginSeriesSettings']) &&
    isset($_POST['devloungeAddContent']) && isset($_POST['devloungeAuthor'])) {
11.                     if (isset($user_email)) {
12.                         $devOptions[$user_email] = $_POST['devloungeAddContent'] .
    "," . $_POST['devloungeAuthor'];
13.                         ?>
14.                             <div class="updated"><p><strong>Settings successfully
    updated.</strong></p></div>
15.                         <?php
16.                         update_option($this->adminUsersName, $devOptions);
17.                     }
18.                 }
19.                 //Get the author options
20.                 $devOptions = $devOptions[$user_email];
21.                 $devOptions = explode(",", $devOptions);
22.                 if (sizeof($devOptions)>= 2) {
23.                     $content = $devOptions[0];
24.                     $author = $devOptions[1];
```

```
25.                 }
26.                 ?>
```

The above code:

- Retrieves the user options (line 7)
- Saved post data (if available) to the database (lines 9 - 18)
- Reads in comma-separated variables for the user.(lines 19-25)

The next bit of code will display the HTML form that is necessary for the user's panel. All the code is doing is displaying the form elements and reading in options that were already retrieved.

PHP:

```
1.  <div class=wrap>
2.  <form method="post" action="<?php echo $_SERVER["REQUEST_URI"]; ?>">
3.  <h2>Devlounge Plugin Series User Options</h2>
4.  <h3>Allow Content Added to the End of a Post?</h3>
5.  <p>Selecting "No" will disable the content from being added into the end of a
    post.</p>
6.  <p><label for="devloungeAddContent_yes"><input type="radio"
    id="devloungeAddContent_yes" name="devloungeAddContent" value="true"
    <?php if ($content == "true") { _e('checked="checked"',
    "DevloungePluginSeries"); }?> /> Yes</label>    <label
    for="devloungeAddContent_no"><input type="radio"
    id="devloungeAddContent_no" name="devloungeAddContent" value="false"
    <?php if ($content == "false") { _e('checked="checked"',
    "DevloungePluginSeries"); }?>/> No</label></p>
7.  <h3>Allow Comment Authors to be Uppercase?</h3>
8.  <p>Selecting "No" will leave the comment authors alone.</p>
9.  <p><label for="devloungeAuthor_yes"><input type="radio"
    id="devloungeAuthor_yes" name="devloungeAuthor" value="true" <?php if
    ($author == "true") { _e('checked="checked"', "DevloungePluginSeries"); }?> />
    Yes</label>    <label for="devloungeAuthor_no"><input
    type="radio" id="devloungeAuthor_no" name="devloungeAuthor" value="false"
    <?php if ($author == "false") { _e('checked="checked"',
    "DevloungePluginSeries"); }?>/> No</label></p>
10. <div class="submit">
11. <input type="submit" name="update_devloungePluginSeriesSettings"
    value="<?php _e('Update Settings', 'DevloungePluginSeries') ?>" /></div>
12. </form>
13.  </div>
14.                 <?php
15.             }//End function printAdminUsersPage()
```

Set up the User's Panel Action

While setting up the administrative panel, we specified a function called **DevloungePluginSeries_ap** that helped initialize the admin panel. We're going to piggy back on this function in order to add in our user's panel.

PHP:

```
1.  //Initialize the admin and users panel
2.  if (!function_exists("DevloungePluginSeries_ap")) {
3.      function DevloungePluginSeries_ap() {
4.          global $dl_pluginSeries;
5.          if (!isset($dl_pluginSeries)) {
6.              return;
7.          }
8.          if (function_exists('add_options_page')) {
9.      add_options_page('Devlounge Plugin Series', 'Devlounge Plugin Series', 9,
    basename(__FILE__), array(&$dl_pluginSeries, 'printAdminPage'));
10.         }
11.         if (function_exists('add_submenu_page')) {
12.             add_submenu_page('profile.php', "Devlounge Plugin Series User
    Options","Devlounge Plugin Series User Options", 0, basename(__FILE__),
    array(&$dl_pluginSeries, 'printAdminUsersPage'));
13.         }
14.     }
15. }
```

On line 12, you can see a line of code that:

- Adds a sub-menu to the **profile.php** page.
- Let users with a user's level greater than or equal to zero access to the user's panel.
- Calls our **printAdminUsersPage** function.

The access level (in this case a 0) is described in more detail at the Users Levels page in the WordPress codex.

Wordpress Plugins and Database Interaction

When you are writing a plugin, you will inevitably have to store variables in a database and retrieve them. Fortunately WordPress makes data retrieval simple with options and a database object. This post will cover storing and retrieving data from a WordPress database.

Storing Data in a Database

There are two main ways to store data in the WordPress database:

1. Create your own table.
2. Use Options

Since most plugins will not require their own database table, I will only cover options. However, the WordPress codex has detailed instructions on how to set up your own table.
WordPress Options

With WordPress options, saving and retrieving data from the database is as simple as a function call. WordPress has four functions for options:

- add_option
- get_option
- update_option
- delete_option

add_option

The **add_option** function accepts four variables, with the name of the option being required. The variables are: add_option($name, $value, $description, $autoload);

This function is beneficial for adding data to the database for retrieval later.

The **$name** variable should be unique so that you don't overwrite someone else's option, or someone else doesn't write over yours.

I usually don't use this function because **update_option** pretty much does the same thing.

get_option

The **get_option** function allows you to retrieve a previously stored option from the database. It accepts only one variable, which is the name of the option to retrieve. The function format is: get_option($option_name);

update_option

The **update_option** function works about the same as the **add_option** function except that it also updates an option if it already exists. I personally like the double functionality of the function and prefer it over **add_option** when storing data to the database.

The function format is: update_option($option_name, $new_value);

delete_option

The delete_option function deletes options from the database. The function format is: delete_option($option_name);

A Code Example

You might recall from previous posts in this series that I stored options in the database as an array. Here is a sample function followed by a brief explanation:

PHP:

```
1.  //Returns an array of admin options
2.  function getAdminOptions() {
3.  $devloungeAdminOptions = array('show_header' => 'true',
4.  'add_content' => 'true',
5.  'comment_author' => 'true',
6.  'content' => '');
7.  $devOptions = get_option($this->adminOptionsName);
8.  if (!empty($devOptions)) {
9.  foreach ($devOptions as $key => $option)
10. $devloungeAdminOptions[$key] = $option;
11. }
12. update_option($this->adminOptionsName, $devloungeAdminOptions);
13. return $devloungeAdminOptions;
14. }
```

On lines 3 - 6, I begin an associative array that will eventually be stored in the WordPress database as an option (line 12). I do this so I don't have to store multiple options (each a database call). This technique helps with code bloat, database queries, and naming collisions with other plugin authors.

The WordPress Database Class

Another powerful method of storing and retrieving data from the WordPress database is using the WordPress Database class object. In a function, a reference to the class would look like:

PHP:

```
1.  function sample_function() {
```

```
2. global $wpdb;
3. }
```

After this variable is referenced, you can access the many useful functions of the **wpdb class**.

For example, say we want to retrieve the total number of comments for our WordPress installation. Here's a function that does that using the WPDB class:

PHP:

```
1. function sample_function() {
2. global $wpdb;
3. $comments = $wpdb->get_row("SELECT count(comment_approved) comments_count FROM $wpdb->comments where comment_approved = '1' group by comment_approved", ARRAY_A);
4. echo $comments['comments_count'];
5. }
```

Here's what the function does:

- On line 2, we add a reference to the **$wpdb** variable.
- On line 3, we call a function inside the **wpdb** class called **get_row**.
- On line 3, we retrieve data from the comments table ($wpdb->comments). We specify that we want the data returned as an associative array (ARRAY_A).
- On line 4, we echo out the result. Since I wanted the results returned as an associative array, I simply call the variable I assigned the results in SQL, which was **comments_count**.

The **wpdb** class is a very large class with a lot of functionality. I suggest heading over the WPDB Class page and looking over what the **wpdb class** is capable of.

Using Javascript and CSS with your Wordpress Plugin

A lot of plugins nowadays are more reliant on JavaScript and Cascading Style Sheets. It is important to separate your JavaScript and CSS into separate files so that the plugin is easier to maintain. This portion of the series will cover how to load JavaScript and CSS files for your plugin.

Add your JavaScript

Your plugin might need to load the Prototype library or perhaps a custom script. This section will show you a few WordPress functions that will allow you to load scripts and avoid script conflicts.

The wp_register_script function

The **wp_register_script** function allows you to register your script for calling and can allow you to set pre-requisites. For example, if your script requires Prototype to have been loaded, you can specify this.

Here are the parameters for **wp_register_script**: wp_register_script($handle, $src, $deps = array(), $ver = false)

- The **handle** is a unique name that you will use to reference your script later. This variable is required.
- The **src** is the absolute source to your JavaScript file. This variable is required.
- The **deps** variable is an array of dependencies. For example, if your script requires prototype, you would list it here. This variable is optional.
- The **ver** variable is a string version of the script. This variable is optional.

Say for example you had a script that was located at: *http://yourdomain.com/wp-content/plugins/your-plugin-directory/js/script.js*

Let's make a few assumptions:

- You want to name the handle 'my_script_handle'.
- Your script depends on the Prototype library.
- Your version is 1.0

You would likely call the function in your plugin code initialization or after a **wp_head** action:

PHP:

```
1. wp_register_script('my_script_handle', 'http://yourdomain.com/wp-content/plugins/your-plugin-directory/js/script.js', array('prototype'), '1.0');
```

The wp_enqueue_script Function

The **wp_enqueue_script** function does the same thing as **wp_register_script** except that the **src** variable is optional. If an **src** is provided, the enqueue function *automatically registers* the script for you, thus making the **wp_register_script** function somewhat unnecessary. However, the **wp_register_script** allows you to register your scripts manually so you can load all of your JavaScripts using one **wp_enqueue_script** function.

If we were to call the same custom script as before, it would be called like this:

PHP:

```
1. wp_enqueue_script('my_script_handle', 'http://yourdomain.com/wp-content/plugins/your-plugin-directory/js/script.js', array('prototype'), '1.0');
```

A JavaScript Example

For the Devlounge Plugin Series plugin, we're going to add in a dummy JavaScript file that will be used in a later post. The purpose of this file is to demonstrate how to use the **wp_enqueue_script** function.

- This file is located at the following location: http://yourdomain.com/wp-content/plugins/devlounge-plugin-series/js/devlounge-plugin-series.js
- The file is dependent upon prototype.
- The version is 0.1

You might recall that in a previous post in this series, we added an action for **wp_head**. The function that was run as a result of that action was called **addHeaderCode**. Let's modify this function to add in our new JavaScript:

PHP:

```
1.  function addHeaderCode() {
2.          if (function_exists('wp_enqueue_script')) {
3.             wp_enqueue_script('devlounge_plugin_series', get_bloginfo('wpurl') . '/wp-content/plugins/devlounge-plugin-series/js/devlounge-plugin-series.js', array('prototype'), '0.1');
4.          }
5.          $devOptions = $this->getAdminOptions();
6.          if ($devOptions['show_header'] == "false") { return; }
7.          ?>
8.  <!-- Devlounge Was Here -->
9.          <?
10.
11.     }
```

The above code does the following:

- The **wp_enqueue_script** function is checked for existence.
- The wp_enqueue_script is called with the **src**, **dependencies**, and **version**.
- We use the **get_bloginfo('wpurl')** to get the location of the WordPress installation and hard-code the rest.

When you go to a post and view-source, the devlounge-plugin-series.js will have loaded as well as the Prototype library, which is conveniently included along with WordPress (versions 2.1.x and up I believe).

Loading in the Cascading Style Sheets

I've added a new style sheet to my styles directory. Here are my assumptions:

- This file is located at the following location: http://yourdomain.com/wp-content/plugins/devlounge-plugin-series/css/devlounge-plugin-series.css
- I specified an ID called **#devlounge-link** in the CSS file.
- **You have added in the following code** at the end of a post: Get the Number of Blog Comments

In the style sheet file, I have added in the following ID:

CSS:

```
#devlounge-link {
    background-color:#006;
    color: #FFF;
}
```

Adding in the style sheet for the plugin is as simple as adding a line to the **addHeaderCode** function:

PHP:

```
function addHeaderCode() {
        echo '<link type="text/css" rel="stylesheet" href="' . get_bloginfo('wpurl') . '/wp-content/plugins/devlounge-plugin-series/css/devlounge-plugin-series.css" />' . "\n";
        if (function_exists('wp_enqueue_script')) {
            wp_enqueue_script('devlounge_plugin_series', get_bloginfo('wpurl') . '/wp-content/plugins/devlounge-plugin-series/js/devlounge-plugin-series.js', array('prototype'), '0.1');
        }
        $devOptions = $this->getAdminOptions();
        if ($devOptions['show_header'] == "false") { return; }
        ?>
<!-- Devlounge Was Here -->
        <?

    }
```

On line 2, I simply echo out a reference to the new style sheet.

Using AJAX with your Wordpress Plugin

More and more plugins are starting to use AJAX techniques. I personally don't see a use for most cases of AJAX, but it may be necessary for your plugin to use AJAX to accomplish a task. This post will show you how to use AJAX with your WordPress plugin.

This post will be building on the last one where we added in a JavaScript and Stylesheet file.

Set Up a new PHP File

The Devlounge Plugin Series plugin has the following directory structure:

- devlounge-plugin-series
  - devlounge-plugin-series.php (main plugin file)
  - js
    - devlounge-plugin-series.js.php
  - css
    - devlounge-plugin-series.css
  - php
    - dl-plugin-ajax.php (new php file for AJAX calls)

Notice I have a **php** extension at the end of my JavaScript file. I'll explain the change a little later in this post.

I've created a new file and placed it in the **php** directory and have called it **dl-plugin-ajax.php**. I have placed the following code inside the file:

PHP:

```php
<?php
if (!function_exists('add_action'))
{
    require_once("../../../../wp-config.php");
}
if (isset($dl_pluginSeries)) {
    $dl_pluginSeries->showComments();
}
?>
```

This code is simple enough and is used solely for AJAX calls. It makes sure that config structure is present so we can call our class object **dl_pluginSeries** and reference other WordPress functions and variables. However, the **showComments** function hasn't been created yet, so that is the next item on our agenda.

Set up the showComments function

The **showComments** function is placed inside our **DevloungePluginSeries** class:

PHP:

```
function showComments() {
        global $wpdb;
        $devloungecomments = $wpdb->get_row("SELECT count(comment_approved) comments_count FROM $wpdb->comments where comment_approved = '1' group by comment_approved", ARRAY_A);
        echo "You have " . $devloungecomments['comments_count'] . " comments on your blog";
    }
```

You might recognize this bit of code from the database interaction post. This function outputs the number of comments made on your blog.

Allow JavaScript to Know Where Your Blog is Located

One pesky thing about AJAX is that the external JavaScript file has no idea where your blog is installed. I get around this by adding a PHP extension to my included JavaScript file so that I can access WordPress functions. Within the **addHeaderCode** function, I changed the code from this:

PHP:

```
if (function_exists('wp_enqueue_script')) {
        wp_enqueue_script('devlounge_plugin_series', get_bloginfo('wpurl') . '/wp-content/plugins/devlounge-plugin-series/js/devlounge-plugin-series.js', array('prototype'), '0.1');
    }
```

to this:

PHP:

```
if (function_exists('wp_enqueue_script')) {
        wp_enqueue_script('devlounge_plugin_seriess', get_bloginfo('wpurl') . '/wp-content/plugins/devlounge-plugin-series/js/devlounge-plugin-series.js.php', array('prototype'), '0.3');
    }
```

The only thing I changed was the version number and added a PHP extension to the end of the JavaScript file.

Now JavaScript has a way of finding out where our blog is for AJAX calls. Let's now set up the JavaScript.

Setting up the JavaScript

The purpose of this script (which is located in **devlounge-plugin-series.js.php**) is to find the blog's URL, call the PHP file, and return a result to the user.

JAVASCRIPT:

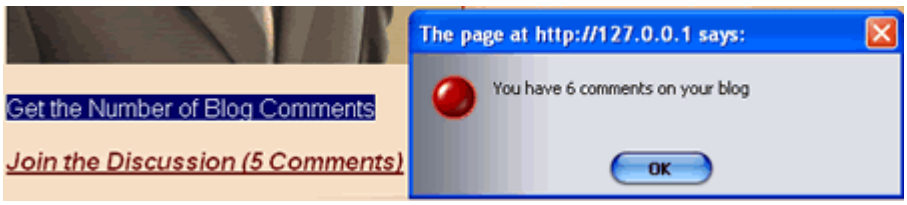
```
1.  <?php
2.  if (!function_exists('add_action'))
3.  {
4.      require_once("../../../../wp-config.php");
5.  }
6.  ?>
7.  Event.observe(window, 'load', devloungePluginSeriesInit, false);
8.  function devloungePluginSeriesInit() {
9.      $('devlounge-link').onclick = devloungePluginSeriesClick;
10. }
11. function devloungePluginSeriesClick(evt) {
12.     var url =  "<?php bloginfo('wpurl') ?>/wp-content/plugins/devlounge-plugin-
    series/php/dl-plugin-ajax.php";
13.     var success = function(t){devloungePluginSeriesClickComplete(t);}
14.     var myAjax = new Ajax.Request(url, {method:'post', onSuccess:success});
15.     return false;
16. }
17. function devloungePluginSeriesClickComplete(t) {
18.     alert(t.responseText);
19. }
```

The above code does the following (keep in mind we are using Prototype):

- Makes sure that config structure is present so we can access WordPress functions and variables.
- After the document has loaded, the **devloungePluginSeriesInit** is called.
- An event is set up for the link you added at the end of a post (line 7). If you forgot, now is the time to add the link in. Simply find a post and add this code to the bottom of it: Get the Number of Blog Comments
- The absolute URL to the PHP file is found (line 12).
- The PHP file is called (line 14).
- The response is outputted to the user (line 18).

The Result

This next step assumes you are at the post where the link was added. When clicking on the link "**Get the Number of Blog Comments**", the script uses AJAX to call a function in the **DevloungePluginSeries** class and returns the result to you in the form of an alert box.



As you can see, I don't have many comments on my local installation.

Releasing and Promoting your Wordpress Plugin

After you have finished writing your awesome WordPress plugin, there are a few things to consider before releasing and promoting your WordPress plugin.

Prior to Release

Try to Follow the Standards

While it isn't required to follow the WordPress coding standards, there are some things in there that will make your life easier. One of the more valuable tips in there is to **never use shorthand PHP**. The reason? Not everybody has shorthand PHP enabled.

So instead of:

PHP:

```
1. <? /*your php code*/ ?>
```

You would have:

PHP:

```
1. <?php /*your php code*/ ?>
```

Make Sure You Have Tested Your Plugin Thoroughly

Find some guinea pigs (er, testers) who would be willing to test your plugin. Technically competent testers are good, but you also want some testers who will represent the average user who knows nothing about programming languages.

It'll be impossible to find every bug, but at least make an effort to put out a stable release.

Make Sure You Have a Readme File

Before you release a plugin into the wild, make sure you at the very least have a Readme file. This file should contain at the very minimum installation instructions for your plugin. For a stricter version of a readme file, check out the WordPress recommendations regarding a Readme file. There's even a groovy Readme file validator.

Set Up a Dedicated WordPress Plugin Page

Ajay D'Souza wrote some recommendations regarding releasing WordPress themes. The advice he gives can also be applied to plugins to an extent.

Make sure you set up a dedicated WordPress Plugin page for your plugin. This page will be the URL that people will go to to find out everything about your plugin. This plugin page should contain the following at a minimum:

- A brief description of your plugin.
- The download location.
- A list of features.
- Install instructions.
- Version history (Changelog).
- Known bugs and/or conflicts.
- Screenshots or a demo (or both).
- Contact or support information (or comments enabled).

The above information will assist you in promoting your plugin, especially the description and feature portion.

Have a Good Folder Structure

I would argue to always include your plugin in a folder. Any files other than the main plugin file should be included in sub-directories. Make sure you zip, gzip, or rar your plugin folder that way it is as easy as possible for people to install your plugin.

Does Your Plugin Require Theme or File Manipulation?

If your plugin requires users to tweak theme settings and/or files, prepare for the onslaught of bug reports and users wanting assistance. I would argue that a good plugin requires absolutely no theme manipulation or file manipulation. An exception to this would be the plugins that add template tags to the WordPress core.

If your plugin does require theme or file manipulation, include detailed examples on your download page and possibly include some examples in your release.

Promoting Your Plugin

After you have your dedicated download page, it is time to start making plugin announcements so people will download your work. The time you spent on your description and features is crucial since you'll be using these for your plugin promotion. Others who link to your plugin will be doing the same.

Promote at Weblog Tools Collection

A very good resource for promoting your plugin is the Weblog Tools Collection news section. Under their Plugin Releases section, you can give details regarding your new plugin.

Promote at the WordPress Plugin Database

The WordPress Plugin Database is another good resource for adding in your plugin. The process for adding your plugin isn't the most straightforward, but there are detailed instructions.

Promote at the Official WordPress Plugin Repository

WordPress itself has offered to host your plugin. You have to meet several requirements before you will be allowed to add your plugin, however. Remember that any publicity is good publicity.

Promote Using Social Networking

Add your plugin to delicious, Digg, and Stumble Upon. Get your friends to help. If your plugin is good enough, the referrals will start coming in.

Promote On Your Own Blog

If your plugin is something that people will notice, use it on your blog. People may start asking what plugin you are using. Word of mouth is a powerful ally, especially in the blogosphere.

Conclusion

You can have the best plugin in the world, but if it isn't released and promoted correctly, very few people will download it. Once you start the promotion process, it is important to listen to feature and bug requests, especially if your plugin is very young. If your plugin doesn't work, or too many people have problems with it, people will be wary of downloading your plugin. It's important to get those bugs fixed and the crucial features added in early. Most of these problems can be solved during testing, but some bugs just don't seem to crop up until after the official release.

The End of the 'How to Write a WordPress Plugin' Series

Thank you for reading the final post in the plugin series. Hopefully this series proved beneficial to you and helped establish a foundation for you to write your own plugins. Thank you very much for reading.

About the site:

Devlounge is a designer and developer resource providing articles, interviews, and exclusive extras such as this article e-book. Please visit http://www.devlounge.net for more content and features just like this by this and other authors. All content is copyrighted © to their respected owners and Devlounge, and may not be reproduced without permission.

A note about Code:

Many code samples used in this series can be downloaded by visiting the specific section page on Devlounge. Be sure to visit the How to Write a Wordpress Plugin main page at http://www.devlounge.net/extras/how-to-write-a-wordpress-plugin to leave feedback, download code, or ask a quick question. You can also dig the article from that page and show your support.