Writing a Plugin

Languages: **English** • Español • 日本語 • 한국어 • Português do Brasil • Русский • ไทย • 中文(简体) • (Add your language)

Introduction

WordPress Plugins allow easy modification, customization, and enhancement to a WordPress blog. Instead of changing the core programming of WordPress, you can add functionality with WordPress Plugins. Here is a basic definition:

WordPress Plugin: A WordPress Plugin is a program, or a set of one or more functions, written in the PHP scripting language, that adds a specific set of features or services to the WordPress weblog, which can be seamlessly integrated with the weblog using access points and methods provided by the WordPress Plugin Application Program Interface (API).

Wishing that WordPress had some new or modified functionality? The first thing to do is to search various WordPress Plugin repositories and sources to see if someone has already created a WordPress Plugin that suits your needs. If not, this article will guide you through the process of creating your own WordPress Plugins.

This article assumes you are already familiar with the basic functionality of WordPress, and PHP programming.

Resources

- To understand how WordPress Plugins work and how to install them on your WordPress blog, see Plugins.
- There is a comprehensive list of articles and resources for Plugin developers, including external articles on writing WordPress Plugins, and articles on special topics, in Plugin Resources.
- To learn the basics about how WordPress Plugins are written, view the source code for well-written Plugins, such as Hello Dolly distributed with WordPress.
- Once you have written your WordPress Plugin, read Plugin Submission and Promotion to learn how to distribute it and share it with others.

Contents

Creating a Plugin

This section of the article goes through the steps you need to follow, and things to consider when creating a well-structured WordPress Plugin.

Names, Files, and Locations

Plugin Name

The first task in creating a WordPress Plugin is to think about what the Plugin will do, and make a (hopefully unique) name for your Plugin. Check out Plugins and the other repositories it refers to, to verify that your name is unique; you might also do a Google search on your proposed name. Most Plugin developers choose to use names that somewhat describe what the Plugin does; for

Home Page
WordPress Lessons
Getting Started
Working with WordPress
Design and Layout
Advanced Topics
Troubleshooting
Developer Docs
About WordPress

Codex Resources

Community portal
Current events
Recent changes
Random page
Help

instance, a weather-related Plugin would probably have the word "weather" in the name. The name can be multiple words.

Plugin Files

The next step is to create a PHP file with a name derived from your chosen Plugin name. For instance, if your Plugin will be called "Fabulous Functionality", you might call your PHP file *fabulous-functionality.php*. Again, try to choose a unique name. People who install your Plugin will be putting this PHP file into the WordPress Plugins directory in their installation (usually *wp-content/plugins/*), so no two Plugins they are using can have the same PHP file name.

Another option is to split your Plugin into multiple files. Your WordPress Plugin must have at least one PHP file; it could also contain JavaScript files, CSS files, image files, language files, etc. If there are multiple files, pick a unique name for a directory and a name of your choice (usually the same) for the main PHP file of your Plugin, such as *fabulous-functionality* and *fabulous-functionality.php*, respectively, put all your Plugin's files into that directory, and tell your Plugin users to install the whole directory under *wp-content/plugins/*. Notice that WordPress installation can be configured for *wp-content/plugins/* directory to be moved, so you must use plugin_dir_path() and plugins_url() for absolute paths and URLs. See: http://codex.wordpress.org/Determining_Plugin_and_Content_Directories for more details.

In the rest of this article, "the Plugin PHP file" refers to the main Plugin PHP file, whether in *wp-content/plugins/* or a sub-directory.

Readme File

If you want to host your Plugin on http://wordpress.org/extend/plugins/, you also need to create a *readme.txt* file in a standard format, and include it with your Plugin. See http://wordpress.org/extend/plugins/about/readme.txt for a description of the format.

Note that the WordPress plugin repository takes the "Requires" and "Tested up to" versions from the *readme.txt* in the stable tag.

Home Page

It is also very useful to create a web page to act as the home page for your WordPress Plugin. This page should describe how to install the Plugin, what it does, what versions of WordPress it is compatible with, what has changed from version to version of your Plugin, and how to use the Plugin.

File Headers

Now it's time to put some information into your main Plugin PHP file.

Standard Plugin Information

The top of your Plugin's main PHP file must contain a standard Plugin information header. This header lets WordPress recognize that your Plugin exists, add it to the Plugin management screen so it can be activated, load it, and run its functions; without the header, your Plugin will never be activated and will never run. Here is the header format:

```
<?php
/*
Plugin Name: Name Of The Plugin
Plugin URI: http://URI_Of_Page_Describing_Plugin_and_Updates
Description: A brief description of the Plugin.
Version: The Plugin's Version Number, e.g.: 1.0
Author: Name Of The Plugin Author
Author URI: http://URI_Of_The_Plugin_Author
License: A "Slug" license name e.g. GPL2
*/
?>
```

The minimum information WordPress needs to recognize your Plugin is the Plugin Name line. The rest of the information (if present) will be used to create the table of Plugins on the Plugin management screen. The order of the lines is not important.

So that the upgrade mechanism can correctly read the version of your plugin it is recommended that you pick a format for the version number and stick to it between the different releases. For example, x.x or x.x.x or xx.xx.xxx

The License slug should be a short common identifier for the license the plugin is under and is meant to be a simple way of being explicit about the license of the code.

Important: file must be in UTF-8 encoding.

License

It is customary to follow the standard header with information about licensing for the Plugin. Most Plugins use the GPL2 license used by WordPress or a license compatible with the GPL2. To indicate a GPL2 license, include the following lines in your Plugin:

```
<?php
/*  Copyright YEAR  PLUGIN_AUTHOR_NAME  (email : PLUGIN AUTHOR EMAIL)

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License, version 2, as
    published by the Free Software Foundation.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*/
?>
```

Programming Your Plugin

Now, it's time to make your Plugin actually do something. This section contains some general ideas about Plugin development, and describes how to accomplish several tasks your Plugin will need to do.

WordPress Plugin Hooks

Many WordPress Plugins accomplish their goals by connecting to one or more WordPress Plugin "hooks". The way Plugin hooks work is that at various times while WordPress is running, WordPress checks to see if any Plugins have registered functions to run at that time, and if so, the functions are run. These functions modify the default behavior of WordPress.

For instance, before WordPress adds the title of a post to browser output, it first checks to see if any Plugin has registered a function for the "filter" hook called "the_title". If so, the title text is passed in turn through each registered function, and the final result is what is printed. So, if your Plugin needs to add some information to the printed title, it can register a "the_title" filter function.

Another example is the "action" hook called "wp_footer". Just before the end of the HTML page WordPress is generating, it checks to see whether any Plugins have registered functions for the "wp_footer" action hook, and runs them in turn.

You can learn more about how to register functions for both filter and action hooks, and what Plugin hooks are available in WordPress, in the Plugin API. If you find a spot in the WordPress code where you'd like to have an action or filter, but WordPress doesn't have one, you can also suggest new hooks (suggestions will generally be taken); see Reporting Bugs to find out how.

Template Tags

Another way for a WordPress Plugin to add functionality to WordPress is by creating custom Template Tags. Someone who wants to use your Plugin can add these "tags" to their theme, in the sidebar, post content section, or wherever it is appropriate. For instance, a Plugin that adds geographical tags to posts might define a template tag function called `geotag_list_states()` for the sidebar, which lists all the states posts are tagged with, with links to the state-based archive pages the Plugin enables.

To define a custom template tag, simply write a PHP function and document it for Plugin users on your Plugin's home page and/or in the Plugin's main PHP file. It's a good idea when documenting the function to give an example of exactly what needs to be added to the theme file to use the function, including the `<?php` and `?>`.

Saving Plugin Data to the Database

Most WordPress Plugins will need to get some input from the site owner or blog users and save it between sessions, for use in its filter functions, action functions, and template functions. This information has to be saved in the WordPress database, in order to be persistent between sessions. There are four (4) methods for saving Plugin data in the database:

1. Use the WordPress "option" mechanism (described below). This method is appropriate for storing relatively small amounts of relatively static, named pieces of data -- the type of data you'd expect the site owner to enter when first setting up the Plugin, and rarely change thereafter.
2. Post Meta (a.k.a. Custom Fields). Appropriate for data associated with individual posts, pages, or attachments. See post_meta Function Examples, add_post_meta(), and related functions.
3. Custom Taxonomy. For classifying posts or other objects like users and comments and/or for a user-editable name/value list of data consider using a Custom Taxonomy, especially when you want to access all posts/objects associated with a given taxonomy term. See Custom Taxonomies.
4. Create a new, custom database table. This method is appropriate for data not associated with individual posts, pages, attachments, or comments -- the type of data that will grow as time goes on, and that doesn't have individual names. See Creating Tables with Plugins for information on how to do this.

WordPress Options Mechanism

See Creating Options Pages for info on how to create a page that will automatically save your options for you.

WordPress has a mechanism for saving, updating, and retrieving individual, named pieces of data ("options") in the WordPress database. Option values can be strings, arrays, or PHP objects (they will be "serialized", or converted to a string, before storage, and unserialized when retrieved). Option names are strings, and they must be unique, so that they do not conflict with either WordPress or other Plugins.

It's also generally considered a good idea to minimize the number of options you use for your plugin. For example, instead of storing 10 different named options consider storing a serialized array of 10 elements as a single named option.

Here are the main functions your Plugin can use to access WordPress options.

```
add_option($name, $value, $deprecated, $autoload);
```

Creates a new option; does nothing if option already exists.

$name
Required (string). Name of the option to be added.

$value
Optional (mixed), defaults to empty string. The option value to be stored.

$deprecated
Optional (string), no longer used by WordPress, You may pass an empty string or null to this argument if you wish to use the following $autoload parameter.

$autoload
Optional, defaults to 'yes' (enum: 'yes' or 'no'). If set to 'yes' the setting is automatically retrieved by the `wp_load_alloptions` function.

```
get_option($option);
```

Retrieves an option value from the database.

$option
Required (string). Name of the option whose value you want returned. You can find a list of the default options that are installed with WordPress at the Option Reference.

```
update_option($option_name, $newvalue);
```

Updates or creates an option value in the database (note that `add_option` does not have to be called if you do not want to use the `$deprecated` or `$autoload` parameters).

$option_name

Required (string). Name of the option to update.

$newvalue

Required. (stringlarraylobject) The new value for the option.

Administration Panels

Assuming that your Plugin has some options stored in the WordPress database (see section above), you will probably want it to have an administration panel that will enable your Plugin users to view and edit option values. The methods for doing this are described in Adding Administration Menus.

Internationalizing Your Plugin

Once you have the programming for your Plugin done, another consideration (assuming you are planning on distributing your Plugin) is *internationalization*. Internationalization is the process of setting up software so that it can be *localized*; localization is the process of translating text displayed by the software into different languages. WordPress is used all around the world, so it has internationalization and localization built into its structure, including localization of Plugins.

Please note that language files for Plugins **ARE NOT** automatically loaded. Add this to the Plugin code to make sure the language file(s) are loaded:

```
        load_plugin_textdomain('your-unique-name', false, basename( dirname( __FILE__ ) ) . '/languages'
);
```

To fetch a string simply use **__('String name','your-unique-name');** to return the translation or **_e('String name','your-unique-name');** to echo the translation. Translations will then go into your plugin's /languages folder.

It is highly recommended that you internationalize your Plugin, so that users from different countries can localize it. There is a comprehensive reference on internationalization, including a section describing how to internationalize your plugin, at I18n for WordPress Developers.

Updating your Plugin

This section describes necessary steps to update your Plugin when you host it on http://wordpress.org/extend/plugins. It especially lists some details regarding the use of Subversion (SVN) with wordpress.org.

Assuming you have already submitted your plugin to the WordPress Plugin Repository, over time you will probably find the need, and hopefully the time, to add features to your Plugin or fix bugs. Work on these changes and commit the changes to the trunk of your plugin, as often as you want. The changes will be publicly visible, but only to the technically-minded people checking out your Plugin via SVN. What other users download through the website or their WordPress Plugin administration will not change.

When you're ready to release a new version of the Plugin:

- Make sure everything is committed and the new version actually works. Pay attention to all WordPress versions your Plugin supports and try to test it with all of them. Don't just test the new features, also make sure you didn't accidentally break some older functionality of the Plugin.
- Change the version number in the header comment of the main PHP file to the new version number (in the trunk folder).
- Change the version number in the 'Stable tag' field of the readme.txt file (in the trunk folder).
- Add a new sub-section in the 'changelog' section of the readme.txt file, briefly describing what changed compared to the last release. This will be listed on the 'Changelog' tab of the Plugin page.
- Commit these changes.
- Create a new SVN tag as a copy of trunk, following this guide.

Give the system a couple of minutes to work, and then check the wordpress.org Plugin page and a WordPress installation with your Plugin to see if everything updated correctly and the WordPress installation shows an update for your Plugin (the update checks might be cached, so this could take some time -- try visiting the 'available updates' page in your WordPress installation).

Troubleshooting:

- The Plugin's page on wordpress.org still lists the old version. Have you updated the 'stable tag' field in the trunk folder? Just creating a tag and updating the readme.txt in the tag folder is not enough!
- The Plugin's page offers a zip file with the new version, but the button still lists the old version number and no update notification happens in your WordPress installations. Have you remembered to update the 'Version' comment in the main PHP file?
- For other problems check Otto's good write-up of common problems: The Plugins directory and readme.txt files

Plugin Development Suggestions

This last section contains some random suggestions regarding Plugin development.

- The code of a WordPress Plugin should follow the WordPress Coding Standards. Please consider the Inline Documentation Standards as well.
- All the functions in your Plugin need to have unique names that are different from functions in the WordPress core, other Plugins, and themes. For that reason, it is a good idea to use a unique function name prefix on all of your Plugin's functions. A far superior possibility is to define your Plugin functions inside a class (which also needs to have a unique name).
- Do not hardcode the WordPress database table prefix (usually "wp_") into your Plugins. Be sure to use the `$wpdb->prefix` variable instead.
- Database reading is cheap, but writing is expensive. Databases are exceptionally good at fetching data and giving it to you, and these operations are (usually) lightning quick. Making changes to the database, though, is a more complex process, and computationally more expensive. As a result, try to minimize the amount of *writing* you do to the database. Get everything prepared in your code first, so that you can make only those write operations that you need.
- Use WordPress' APIs instead of using direct SQL where possible. For example, use `get_posts()` or `new WP_Query()` instead of `SELECT * FROM {$wpdb->prefix}_posts`.
- Use the existing database tables instead of creating new custom tables if possible. Most use-cases can be accomplished with custom post types and meta data, custom taxonomy and/or one of the other standard tables and using the standard tables provides a lot of UI and other functionality *"for free."* Think very carefully before adding a table because it adds complexity to your plugin that many users and site builders prefer to avoid.
- SELECT only what you need. Even though databases fetch data blindingly fast, you should still try to reduce the load on the database by only selecting that data which you need to use. If you need to count the number of rows in a table don't `SELECT * FROM`, because all the data in all the rows will be pulled, wasting memory. Likewise, if you only need the post_id and the post_author in your Plugin, then just `SELECT` those specific fields, to minimize database load. Remember: hundreds of other processes may be hitting the database at the same time. The database and server each have only so many resources to spread around amongst all those processes. Learning how to minimize your Plugin's hit against the database will ensure that your Plugin isn't the one that is blamed for abuse of resources.
- Eliminate PHP errors in your plugin. Add `define('WP_DEBUG', true);` to your *wp-config.php* file, try all of your plugin functionality, and check to see if there are any errors or warnings. Fix any that occur, and continue in debug mode until they have all been eliminated.
- Try not to echo <script> and <style> tags directly - instead use the recommended wp_enqueue_style() and wp_enqueue_script() functions. They help eliminate including duplicate scripts and styles as well as introduce dependency support. See posts by the following people for more info: Ozh Richard, Artem Russakovskii, and Vladimir Prelovac.

External Resources

- Anatomy of a WordPress Plugin Featuring Hello Dolly (09March11)
- Top 10 Most Common Coding Mistakes in WordPress Plugins (11SEP09)
- WordPress 2.0.3: Nonces (Secure your forms with nonces) (02JUN06)
- Simplified AJAX For WordPress Plugin Developers using Jquery(10APR08)
- "Desenvolvendo Plugins para WordPress" by Rafael Dohms (in Brazilian Portuguese) (10MAR08)
- 12 part "How to Write a Wordpress Plugin" at DevLounge.net by Ronald Huereca
- How to create WordPress Plugin from a scratch (9AUG07)
- HookPress, a plugin that enables extending WordPress in languages other than PHP via webhooks. (26SEP09)
- How To Include CSS and JavaScript Conditionally And Only When Needed By The Posts (13JAN10)
- Demonstrating how to use the Settings API, WP_Http, and Pseudo-cron (01MAR10)
- Make WordPress Plugins - official blog with information for plugin authors (25Jun12)
- WordPress Generator - GenerateWP provides user-friendly tools for developers to create advanced systems built on WordPress. (15JAN13)
- How to write a WordPress Plugin - A simple how-to guide to writing a class based WordPress plugin with a settings page, custom

post type and metaboxes that separates business and template logic in an easy to understand way. (31Jan13)

Categories: Plugins | WordPress Development | UI Link