Table of Contents

Chapter 5

Plugin Development

Having written a few plugins, I have learned a lot about writing them well and in a secure way. So some of what I have learned over the last few years will be imparted to you here. I'm still learning and gaining new insights into what a useful plugin should look like and how it should respect its environments on both the admin and public sides.

Let's start with a few of the ground rules and then look at the coding aspects of this process. Keep in mind there are two major development bases. The first, WordPress, has a lot of its built-in functions that you should consider first. The second is PHP itself, the programming language on which WordPress is based. The PHP ecosystem can provide generalized solutions you can adapt for use on your site. These are usually integrations with other services, APIs, and frameworks.

This chapter is our first more profound look into programming—more so than the rest of the book. If you have no interest in writing code, perhaps this chapter will whet your appetite. It's often a desire to improve a product in which you see flaws. It may follow that as you see flawed plugins, you can fix the code yourself, or create a new plugin from scratch which solves the same issue but more logically and efficiently.

Preparation, Rules, Best Practices

Next, we want to look at practices we should follow when building a plugin to ensure that it is maintainable, secure, and doesn't break your site. In list format, they are as follows.

1. Don't Mess with the WordPress Core.
2. Keep your code separate from other plugins you may be developing, even if they are complementary.
3. Use the WordPress API functions whenever possible; they save time and are native.
4. Prefix your code and tables with unique identifiers to avoid naming collisions.
5. Use object-oriented programming (OOP) whenever possible.
6. Follow secure coding guidelines.
7. Don't use sessions if possible.
8. Use WordPress functions instead of PHP functions.
9. Always use the WordPress Database API when interacting with the database.

Rule One: Don't Mess With The WordPress Core!

Emblazon this rule on your forehead and heart! The core of WordPress is the foundation of the whole website structure well beyond your plugin. If you ever make a change to the core files, you endanger your plugin work and your website's very foundation. Not only do you jeopardize your site, but on the off chance that your core code alterations work out, the risk of instability returns when a new version of WordPress is released. When fixes or enhancements are added to the core codebase of WordPress, you will be out of step with it and, in fact, may even conflict with it. Updates will be a pain if they can be done at all, and your plugin may have to be adjusted with much work to get it realigned. **So, don't mess with the WordPress core!**

Rule Two: Keep Your Code Separate

Keep your plugin code separate from other plugins and keep theme customizations in a distinct child theme. This habit helps with potential conflicts and keeps your code from being dependent on code that may not be defined or installed by a user. It allows each plugin to be 100% independent and helps ensure it can stand and execute independently. Even if you are creating a plugin that augments another one, you should still keep the source code of each one separate as there is no guarantee an end-user will install the same code you may be depending on. Additionally, if you are creating a plugin dependant on another plugin (e.g., one that extends WooCommerce), be sure to check this out as a way to test a dependant plugin exists[1]. Where your code does depend on another plugin or library, document that dependency.

Rule Three: Use WordPress API Functions

Using the functions already provided by WordPress saves time and effort because you are using tried and tested solutions. You are not reinventing the wheel with code that needs to be reviewed and vetted. It saves you time and mental processing cycles by letting you focus on solving the problem at hand. See the list of APIs here[2].

Here are a few of the more common APIs you may like to use in your plugins:

Dashboard Widgets API—if your plugin adds content (notice information) to the admin dashboard Database API—if you are directly interfacing with the database REST API—lets you make some or all WordPress data (posts, pages, etc.) available to external platforms Options API—if you want your plugin options controlled alongside other admin options Shortcode API—if you plan to make shortcodes a part of your plugin Widgets API—allows your plugin to make configurable widgets available to the theme.

Rule Four: Prefix Your Code

Prefixing your function and class names helps encapsulate your code from other plugins that may also be installed on the same WordPress site. Many plugins interact with the database. If you had a function called `save_to_db()` and another plugin defines the same function, there is ambiguity. One or both plugins can fail—bringing the entire site to a stand-still.

[1] dependant plugin exists: https://phpa.me/devinvinson-468
[2] list of APIs here: https://codex.wordpress.org/WordPress_APIs

This is an issue within the underlying language of PHP; when it encounters a call to a function with an already used name, it comes to a complete halt and emits a fatal error. For example, if your plugin was named “Plugin Services Manager,” prefix your save to database function with `psm_`. Your function’s full name would then be: `psm_save_to_db()`.

This advice goes back to rule two. You could copy well-tested code from a previously developed plugin and therefore be in self-conflict. Although it may not seem efficient to have potentially duplicated code, it is best, in the long run, to be as separated and ensconced as possible with your plugin code. If you find that you use the same functions across plugins, consider making a “base” plugin that your other plugins depend on.

Additionally, you can use PHP’s native syntax for namespaces and object-oriented programming to prevent name collisions. Using namespaces prevents naming clashes in large applications by segmenting class and function names. You can read up on namespaces in PHP[3] in the online documentation.

Rule Five: Use OOP Whenever Possible

This advice is a general practice for anything related to PHP code. Object-oriented programming is a better approach to writing and organizing code. It makes your code more reusable and adaptable. It also helps with the separation of coding concerns, as already pointed out above. OOP allows for the encapsulation of code into blueprints called classes, which define how objects of a particular class behave. These blueprints house both properties (variables) and methods (functions) specific to that object. Once instantiated, the object is treated as one item with defined behaviors. For further reading on OOP check out these resources:

- Simplifying WordPress’s functions.php with OOP[4]
- Introduction to Object-Oriented PHP for WordPress Developers[5]

Rule Six: Follow Secure Coding Practices

Even in this modern age of software development, we need to reiterate the importance of secure code. Keeping your site safe from malicious actors is an ongoing process, but it can save you time and effort by minimizing the risk that your site is taken over, defaced, or used to steal user data. Here are some general guidelines for writing secure code.

1. Secure and filter input. You should never trust any information coming from an outside/unknown source, especially if your plugin allows for things like public

[3] *namespaces in PHP: https://php.net/language.namespaces.rationale*
[4] *Simplifying WordPress’s functions.php with OOP: https://phpa.me/wsmith-wp-oop*
[5] *Introduction to Object-Oriented PHP for WordPress Developers: https://phpa.me/wpshout-oop-course*

comments. If you're expecting a state abbreviation, US phone number, or zip code, validate that the input looks like what you expected.

2. Escape all output. Never display user-submitted data without escaping it first. Doing so prevents cross-site scripting (XSS) attacks where someone tries to add HTML or Javascript code to your page. (Cross-Site Scripting) site attacks.
3. Confirm only authorized users have expected access levels granted to your plugin. Always ensure your site contributors have roles with appropriate levels of access. You should never be in danger of being "hacked" by an authorized user with unnecessary access. There's no need to be hacked by someone who has authorized access to your site. Be sure to give your site contributors appropriate access roles to complete their tasks and no more.
4. Protect database insertions with prepared statements and filtered data. Clean data going into your database is always preferred and prevents frequent attacks. Use prepared statements to avoid SQL injection attacks.

WordPress has several functions that can help secure your code and data; be sure to employ them when and where needed. Look here for details on this security topic[6].

Rule Seven: Minimize Using Sessions If Possible

Sessions should only be used where really needed. WordPress.org recommends that if you must use a session, you at least encapsulate it within a function. Using sessions can conflict with server-based caching features that are often set up on commercial hosting platforms. Using PHP functions like `session_start()`, `ob_start()`, and `ob_end_flush()` can conflict with products like Varnish and NGINX that cache content on a site-wide basis. This can cause the host to not work as expected and may get your plugin banned from the host as a result. It may be a non-issue issue if you are running your hosting platform, but the nature of plugin creation is to allow the masses to use them, so it is better not to use code caching methods.

Rule Eight: Use WordPress Functions Instead Of PHP Functions

Rule eight is a similar point to the one above about APIs. WordPress has many functions that mimic or improve upon native PHP functions. There are often specific situations within WordPress that the native PHP function does not account for. The WordPress replacement function takes these kinds of issues into account. Therefore, your code benefits from these improvements if you use the WordPress equivalent. One example of this is the augmentation of PHP's `strip_tags()` function into WordPress' `wp_strip_all_tags()`[7] function. Read the latter's documentation for an example.

[6] *security topic: https://phpa.me/wpdev-plugins-security*
[7] `wp_strip_all_tags()`*: https://phpa.me/wpdev-wp-strip-all-tags*

Rule Nine: Always Use The WordPress Database API

WordPress has its database class accessed through `$wpdb`; use it and its methods at all times. Also, use prepared statements when building your SQL commands to avoid making your code vulnerable to SQL injections. See the `$wpdb` class documentation here[8] for usage details.

Further Guidelines

That covers the best practices of plugin creation, in general, and at a high level. Be sure to review all of the guidelines[9] for plugin development.

> *NOTE: Your development platform should be separate from any live WordPress installations while building your plugins. Some major development projects even have another platform, often called a staging site to perform testing.*

Example Plugin

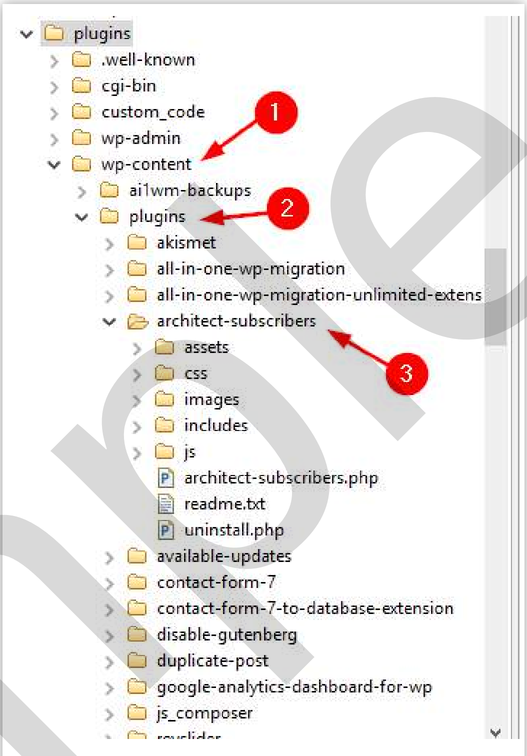
Let's carry on with building an example plugin over the course of this chapter following the best practice guidelines as we progress. First, let's discuss what this plugin will do. It will be basic in its functionality and features. We also want to show how to interact with the front and backend of WordPress—the public and admin areas, respectively. We review processes like connecting to the database, plugin installation, activation, and removal of the plugin. With these basic concepts and integration points, you can get started and create useful plugins for the WordPress world.

Our plugin allows a website visitor to provide their name and email address on a sidebar-based form. Upon opt-in confirmation, we store any collected email addresses in the database in our own table. We can use the contact information to announce when we publish new blog posts on the website. The admin pages allow for turning on or off this new post announcement process and manual data entry. The plugin can also list all the collected data and delete selected emails from the admin area. The email owner can unsubscribe at any time via a link embedded in the emails that they receive via the plugin. We're keeping this a simple example plugin, as previously mentioned. Therefore, we are not creating any shortcode compatibility or adding extra feature add-ons like editing content on the admin side. This code will be free to use and to extend.

[8] here: https://phpa.me/wpdev-wpdb
[9] guidelines: https://phpa.me/wpdev-plugin-practices

Basic Files And Folder Layouts

Let's get started with the basic folder structure of a typical WordPress plugin; see Figure 5.1 as a structural example.



```
plugins
  .well-known
  cgi-bin
  custom_code
  wp-admin
  wp-content
    ai1wm-backups
    plugins
      akismet
      all-in-one-wp-migration
      all-in-one-wp-migration-unlimited-extens
      architect-subscribers
        assets
        css
        images
        includes
        js
        architect-subscribers.php
        readme.txt
        uninstall.php
      available-updates
      contact-form-7
      contact-form-7-to-database-extension
      disable-gutenberg
      duplicate-post
      google-analytics-dashboard-for-wp
      js_composer
```

Figure 5.1.

The code should be in its own folder under the `wp-content/plugins` folder (3). If your plugin name is multi-worded, use hyphens between the words and keep the folder name all lowercase to avoid any potential operating system issues. Our plugin will be called `architect-subscribers`. You should try to name the plugin as descriptively as possible to give users an idea of who created the plugin and what it is meant to do. The primary plugin file name should be the same as the folder name but with a `.php` extension (`architect-subscribers.php`). Regardless of whether you plan to make your plugin commercially available, it's always a good idea to have a `readme.txt` file accompanying the plugin to explain what it is, any dependencies required, and configuration notes in more detail. Read here for some guidelines on the content of the readme.txt file[10].

Plugin folder structure is flexible but should include the following folders at the very least.

- `assets`—screenshots of your plugin
- `css`—where your CSS (cascading style sheets) are stored
- `images`—where your plugin images and icons are stored
- `includes`—where your plugin code files go
- `js`—where any JavaScript or jQuery code is stored

[10] readme.txt file: https://phpa.me/wp-readme-works

Next, we build the code for the main PHP plugin file. This file sets up most of the plugin's environment and establishes the main menu on the admin side. However, before we build the menu structure, we need to register the plugin within the WordPress ecosystem. To do this, we include the plugin metadata at the beginning of the main file, which names the plugin, gives the contact references, and establishes the license under which the plugin code is released to the public. Listing 5.1 shows the commented code block, which determines all of this information.

Listing 5.1.

```php
<?php
/*
 Plugin Name: Architect Subscribers
 Plugin URI:  https://paladin-bs.com/plugins/
 Description: Plugin sample for teaching plugin Development.
 Author:  Peter MacIntyre
 Version: 1.2
 Author URI:  https://paladin-bs.com/peter-macintyre/
 Details URI: https://paladin-bs.com
 License: GPL2
 License URI: https://www.gnu.org/licenses/gpl-2.0.html

 Architect Subscribers is free software: you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation, either version 2 of the License, or
 any later version.

 Architect Subscribers 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.

 See License URI for full details.
 */
```

Here we give the plugin its name, its description, its version number and author, its URI for where it can be found, and its license. Generally, the license is the GPL2 license; we also provide the URI for where this license can be further explored. Having this block of comments at the beginning of the file registers the plugin on the list of installed (active or inactive) plugins within the WordPress admin area. If you fail to format these settings properly, you risk not having the plugin display on this list. Formatting accuracy is essential. Figure 5.2 shows the results of proper formatting by listing the plugin on the plugins page.

Index