

Table of Contents

Foreword	VII
1. Planning a Website	1
Overview	2
Why WordPress?	3
Technical Concerns	3
Operational Considerations	6
2. Development Environment Requirements	11
HTML and CSS	11
Overview	12
Server Platforms Environments	14
WordPress Install	15
3. CSS Crash Course	21
HTML	22
Style Sheets	25
Responsive CSS	36
Conclusion	56
4. The Gutenberg Editor	57
The Block Editor	58
Gutenberg's Origin	65
Building A Block	67
Reusing A Block	72
When To Use Gutenberg	74

TABLE OF CONTENTS

5. Plugin Development	75
Preparation, Rules, Best Practices	76
Example Plugin	80
Basic Files And Folder Layouts	81
Collecting And Saving Data	85
Listing Collected Data	90
Admin Menu Creation	95
Admin Page Development	98
Form Management	98
Creating A Widget	104
Creating A Dashboard Widget	113
Troubleshooting And Debug Tools	114
Installation And Update Processes	115
Distributing Your Plugins	119
Summary	121
6. Child Themes	123
Why Build A Child Theme?	124
Customizing The Twenty Nineteen Theme	125
Customizing Themes With The Child Themes Strategy	128
Template Hierarchy	128
Quick Reference	134
7. Internationalization	137
Translation Functions	138
Preparing Your Code	140
Translating A Settings Page	141
The Loco Translate Plugin	143
Viewing The Results	146

8. The Best Plugins—Part One (Simple Tasks)	147
How To Install And Activate A Plugin	149
Akismet	151
Available Updates	151
Testimonial Rotator	152
Custom Sidebars	154
Store Locator Plus	154
Duplicate Post	156
Minimal Coming Soon & Maintenance Mode	156
Add From Server	157
Pop-up Maker	158
Simple Links	159
WP Super Cache And Autooptimize	160
Smush	162
WP Content Copy Protection & No Right Click	163
GDPR Cookie Consent	164
WP Rollback	165
Custom Login	166
Disable Gutenberg	167
Wicked Folders	167
Force Login	168
9. The Best Plugins—Part Two (Advanced Tasks)	171
Wordfence Security	172
Google Analytics Dashboard For WP	174
WP Forms	175
Revolution Slider (\$)	178
WooCommerce	179

Yoast SEO	183
Ultimate Member	186
All-in-One WP Migration	187
HyperDB	188
Adminimize	189
Custom Post Types	190
Posts 2 Posts	192
RCCP-Free	196
10. Other Resources	199
Themes	200
Web Resources	200
Troubleshooting Assistance	202
Finding a Great Host	203
The Community	204
Index	205

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.

5. PLUGIN DEVELOPMENT

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

5. PLUGIN DEVELOPMENT

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 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>

5. PLUGIN DEVELOPMENT

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.

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].

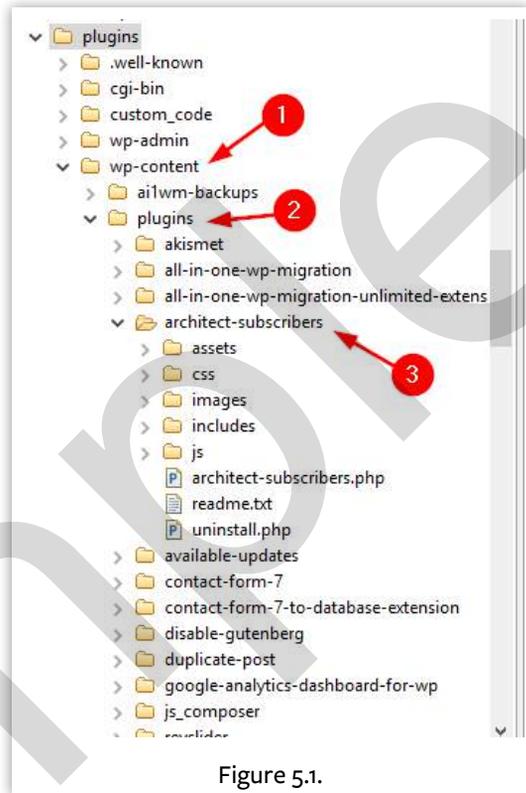


Figure 5.1.

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>

5. PLUGIN DEVELOPMENT

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.

```
1. <?php
2. /*
3.  Plugin Name: Architect Subscribers
4.  Plugin URI: https://paladin-bs.com/plugins/
5.  Description: Plugin sample for teaching plugin Development.
6.  Author: Peter MacIntyre
7.  Version: 1.2
8.  Author URI: https://paladin-bs.com/peter-macintyre/
9.  Details URI: https://paladin-bs.com
10. License: GPL2
11. License URI: https://www.gnu.org/licenses/gpl-2.0.html
12.
13. Architect Subscribers is free software: you can redistribute it and/or modify
14. it under the terms of the GNU General Public License as published by
15. the Free Software Foundation, either version 2 of the License, or
16. any later version.
17.
18. Architect Subscribers is distributed in the hope that it will be useful,
19. but WITHOUT ANY WARRANTY; without even the implied warranty of
20. MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
21. GNU General Public License for more details.
22.
23. See License URI for full details.
24. */
```

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

A

- access, 5–6, 8, 15, 17, 79, 90, 95, 130, 149, 168–70, 173, 175–76, 186, 189–90, 196
 - administration, 8
 - administrator-level, 98
 - appropriate levels of, 79, 98
 - authorized, 79
 - blocking, 169
 - limiting, 186
- accessibility, 34, 62, 74
 - technical, 74
 - visual, 60
- Add from Server, 157
- Adminize, 190
- Akismet, 16, 148, 151
 - plugin, 9, 165
- All-in-One WP Migration, 20, 150, 187
- Amazon S3, 187
- Apache Web Server, 12, 14
- APIs, 76–77, 79
 - common, 77
 - plugin Widgets, 77
 - telephony, 196
- Autooptimize, 160–62
- Available Updates, 151

B

- backups, 20, 187–88
 - complete, 166
 - multiple, 188
 - restorable, 188
- block
 - context, 59–60
 - creation, 67
 - image, 61–63
 - meta, 67
 - multiple, 73
 - new, 61, 73
 - paragraph, 58, 60–61
 - reCAPTCHA, 112
 - regular, 74
 - reusable, 59, 72–74
 - script, 68–69
- block.js, 68–69

C

- cache, 13, 29, 161, 202
 - bust, 29
 - page output, 160
- Cascading Style Sheets. See CSS
- CDN (Content Delivery Network), 161
- community, 24, 65, 67, 200–201, 204
- conferences, 25, 201–2

10. INDEX

container, 12, 39–43, 45–56
 flex, 45–46, 48–49, 51
 grid, 51
Content Editor, 65, 67, 154, 192
CPT plugin. See Custom Post Types
cross-site scripting, 79
CSS
 advanced selectors, 34–35
 code, 32, 84, 104
 flex, 45–52, 54–55
 flexbox, 39, 43–45, 48, 56
 inline, 31
 parent theme's, 126
 responsive, 36–37, 39, 41, 43, 45, 47, 49, 51, 53, 55
 rules, 25, 29, 32, 34, 36
Custom Post Types, 131, 167, 186, 190–91, 193

D

database, 16–17, 20, 76–77, 79–80, 85, 99, 108, 118, 189
 backup, 187
 named, 17
 servers, 12–13, 188
devices, 3, 25, 36, 179, 205
 mobile, 3, 159, 161, 179

E

editor, 58, 61, 65, 67–69, 167, 192
 block, 58–59, 61, 63, 66, 74
 Classic, 60, 66, 74, 131, 192
 Gutenberg, 57–58, 60, 62, 64, 66, 68, 70–72, 74
environment, 13–16, 67, 75
 development, 13–15, 17, 74
 local, 15
 production, 13, 172
 staging, 15

F

file
 backup, 187–88
 cached, 29, 161
 core, 76
 functions, 126
 language, 140, 144
 translation, 140, 143–45
 uninstall.php, 118–19
Force Login, 168–69
FTP, 17, 19, 149, 157
 credentials, 150
 secure, 16

G

GDPR, 8, 118, 164
 compliance, 164
General Data Protection Regulation. See GDPR
Git, 19, 29
GNU General Public License, 82, 120, 125
Google Analytics Dashboard, 7, 174

Google Maps, 154
 Google ReCAPTCHA, 102, 109, 176
 Site Key, 101–2
 Grid Layout, 39–40, 43–44
 example, 41–43
 Gutenberg, 57–59, 65–67, 72, 74, 131, 167, 192
 Disable, 167

I

Internationalization, 137–38, 140, 142, 144, 146

J

JavaScript, 22–23, 67–69, 81, 84, 161, 202
 modern, 26

M

MAMP, 14
 MariaDB, 12
 Matomo, 7
 menu, 6, 36, 72, 95–98, 114, 163, 189–91
 item, 95, 98, 149–50, 157, 174–76, 180, 187
 main, 82, 98
 pop-up, 163
 slug, 96–97
 title, 96–97
 Mullenweg, Matt, 65, 204
 MySQL, 12–14, 202
 server, 15
 settings, 17

O

object-oriented programming (OOP), 76, 78, 108

P

payment processing gateway, 182
 PayPal, 6, 180, 182
 PHP
 ecosystem, 76
 Functions, 76, 79, 84
 settings, 187
 version, 203
 plugin
 code, 13, 77–78, 82–83, 93, 118, 120
 default, 15–16, 151
 development, 75–76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104
 distribute, 119
 ecommerce, 148
 example, 80, 138
 installation, 80, 149
 installed, 143, 150, 152, 165
 sample, 113, 138, 140
 slider, 178
 posts
 context, 59–60
 current, 64, 195
 types, 130–31, 134, 167, 191–94
 POT file, 144
 master, 145

R

reCAPTCHA, 101–2, 105–6, 109–12, 173, 176

10. INDEX

S

Search Engine Optimization. See SEO
security, 8, 18, 85, 138, 152, 174, 188
 practices, 68
 risk, 9
 SQL injection attacks, 79
SEO, 6–7, 171, 183–84
 content, 183, 185
 techniques, 7
shortcodes, 58, 67, 72, 77, 155, 159–60, 177, 179
 advanced, 58, 67
Simple Links, 159
site
 blog, 151
 performance, 188–89
 security, 8, 172
 staging, 80, 164, 166
SMS, 197
style sheets, 25, 27, 29, 31–33, 35–36, 125–27
 base, 29
 separate, 36
style tags, 29–31

T

taxonomies, 130, 134–35, 184
template, 58, 73–74, 85, 117–18, 125–26, 128,
131, 133–35, 144, 178
 file, 128, 131, 144–45, 154
 hierarchy, 128–29, 131, 133
 page, 131, 133
theme, 2, 5, 60, 62, 65, 74, 77, 121, 123–28,
131, 135, 165, 188, 195, 200–202
 active, 128
 child, 12, 121, 123–28, 130, 132, 134, 181, 195
 commercial, 178, 200, 203
 default, 5, 124
 customize, 127–28
 name, 125, 127
 original, 124–25
 parent, 121, 124, 126–27, 133
translations, 138–39, 141–46

U

updates, 17–19, 29, 59, 74, 76, 100, 103,
115–16, 118, 120, 128, 142–43, 148–49,
173–74, 207
 blocking, 124
 manually, 193
 minor, 18

V

VueJS, 31

W

WAMP, 14

web server, 12–13, 15, 29, 161

Wicked Folders, 167

widget, 104–9, 111, 113, 152–54, 180–81, 190

 dashboard, 113, 120

 shopping cart, 154

 signup, 197

WooCommerce, 6, 77, 114, 128, 148, 167,
179–83, 190

Wordfence, 8, 16, 18, 172–74

WordPress

 admin area, 82, 97, 165, 197

 codex, 5, 129

 core, 76–77, 190, 204

 documentation, 67, 125, 127

 installations, 19–20, 97, 151, 197

 multisite instances, 189

 plugins, 57, 81, 113, 127, 201

 themes, 3, 123, 125, 200

 UI, 154, 190–91

 version, 18, 115–16, 168

WP Content Copy Protection, 163

WP Forms, 175, 177

WP Rollback, 165

WP Super Cache, 160–61

Y

Yoast SEO, 6, 183, 185

Z

ZIP file, 120, 150