

SECOND EDITION



Professional
WordPress®
Plugin Development

Brad Williams, Justin Tadlock, John James Jacoby

Table of Contents

[COVER](#)

[FOREWORD](#)

[INTRODUCTION](#)

[WHO THIS BOOK IS FOR](#)

[WHAT YOU NEED TO USE THIS BOOK](#)

[WHAT THIS BOOK COVERS](#)

[HOW THIS BOOK IS STRUCTURED](#)

[CONVENTIONS](#)

[SOURCE CODE](#)

[ERRATA](#)

[1 An Introduction to Plugins](#)

[WHAT IS A PLUGIN?](#)

[AVAILABLE PLUGINS](#)

[ADVANTAGES OF PLUGINS](#)

[INSTALLING AND MANAGING PLUGINS](#)

[SUMMARY](#)

[2 Plugin Framework](#)

[REQUIREMENTS FOR PLUGINS](#)

[BEST PRACTICES](#)

[PLUGIN HEADER](#)

[DETERMINING PATHS](#)

[ACTIVATE/DEACTIVATE FUNCTIONS](#)

[UNINSTALL METHODS](#)

[CODING STANDARDS](#)

[SUMMARY](#)

[3 Dashboard and Settings](#)

[ADDING MENUS AND SUBMENUS](#)

[PLUGIN SETTINGS](#)

[THE OPTIONS API](#)

[THE SETTINGS API](#)

[KEEPING IT CONSISTENT](#)

[SUMMARY](#)

[4 Security and Performance](#)

[SECURITY OVERVIEW](#)

[USER PERMISSIONS](#)

[NONCES](#)

[DATA VALIDATION AND SANITIZATION](#)

[FORMATTING SQL STATEMENTS](#)

[SECURITY GOOD HABITS](#)

[PERFORMANCE OVERVIEW](#)

[CACHING](#)

[TRANSIENTS](#)

[SUMMARY](#)

[5 Hooks](#)

[UNDERSTANDING HOOKS](#)

[ACTIONS](#)

[FILTERS](#)

[USING HOOKS FROM WITHIN A CLASS](#)

[USING HOOKS WITH ANONYMOUS FUNCTIONS](#)

[CREATING CUSTOM HOOKS](#)

[FINDING HOOKS](#)

[SUMMARY](#)

[6 JavaScript](#)

[REGISTERING SCRIPTS](#)

[ENQUEUEING SCRIPTS](#)

LIMITING SCOPE
LOCALIZING SCRIPTS
INLINE SCRIPTS
OVERVIEW OF BUNDLED SCRIPTS
POLYFILLS
YOUR CUSTOM SCRIPTS
jQuery.
BACKBONE/underscore
REACT
SUMMARY

7 Blocks and Gutenberg
WHAT IS GUTENBERG?
TOURING GUTENBERG
PRACTICAL EXAMPLES
TECHNOLOGY STACK OF GUTENBERG
“HELLO WORLD!” BLOCK
WP-CLI SCAFFOLDING
CREATE-GUTEN-BLOCK TOOLKIT
BLOCK DIRECTORY
SUMMARY

8 Content
CREATING CUSTOM POST TYPES
POST METADATA
META BOXES
CREATING CUSTOM TAXONOMIES
USING CUSTOM TAXONOMIES
A POST TYPE, POST METADATA, AND TAXONOMY PLUGIN
SUMMARY

9 Users and User Data

WORKING WITH USERS

ROLES AND CAPABILITIES

LIMITING ACCESS

CUSTOMIZING ROLES

SUMMARY

10 Scheduled Tasks

WHAT IS CRON?

SCHEDULING CRON EVENTS

TRUE CRON

PRACTICAL USE

SUMMARY

11 Internationalization

INTERNATIONALIZATION AND LOCALIZATION

CREATING TRANSLATION FILES

SUMMARY

12 REST API

WHAT THE REST API IS

WHAT YOU CAN DO WITH THE REST API

ACCESSING THE WORDPRESS REST API

THE HTTP API

WORDPRESS' HTTP FUNCTIONS

BRINGING IT ALL TOGETHER

SUMMARY

13 Multisite

TERMINOLOGY

ADVANTAGES OF MULTISITE

ENABLING MULTISITE IN WORDPRESS

MULTISITE FUNCTIONS

[DATABASE SCHEMA](#)

[QUERY CLASSES](#)

[OBJECT CLASSES](#)

[SUMMARY](#)

[14 The Kitchen Sink](#)

[QUERYING AND DISPLAYING POSTS](#)

[SHORTCODES](#)

[WIDGETS](#)

[DASHBOARD WIDGETS](#)

[REWRITE RULES](#)

[THE HEARTBEAT API](#)

[SUMMARY](#)

[15 Debugging](#)

[COMPATIBILITY](#)

[DEBUGGING](#)

[ERROR LOGGING](#)

[QUERY MONITOR](#)

[SUMMARY](#)

[16 The Developer Toolbox](#)

[CORE AS REFERENCE](#)

[PLUGIN DEVELOPER HANDBOOK](#)

[CODEX](#)

[TOOL WEBSITES](#)

[COMMUNITY RESOURCES](#)

[TOOLS](#)

[SUMMARY](#)

[INDEX](#)

[END USER LICENSE AGREEMENT](#)

List of Tables

Chapter 3

[TABLE 3-1: List of Core Sections and Fields](#)

Chapter 12

[TABLE 12-1: Main HTTP Status Codes](#)

[TABLE 12-2: HTTP Status Code Classes](#)

[TABLE 12-3: Default Settings of wp_remote Functions Optional Parameters](#)

List of Illustrations

Chapter 1

[FIGURE 1-1: Loading a page in WordPress](#)

[FIGURE 1-2: Plugins menu](#)

[FIGURE 1-3: Install Now button](#)

[FIGURE 1-4: Types and statuses for plugins](#)

Chapter 3

[FIGURE 3-1: Custom registered menu](#)

[FIGURE 3-2: Submenus](#)

[FIGURE 3-3: Submenu labeled PDEV Settings](#)

[FIGURE 3-4: Plugin management page](#)

[FIGURE 3-5: Error message](#)

[FIGURE 3-6: Section appended](#)

[FIGURE 3-7: Singular field](#)

[FIGURE 3-8: Heading levels](#)

[FIGURE 3-9: Dashicons](#)

[FIGURE 3-10: Dismissible notices](#)

[FIGURE 3-11: WordPress-styled button](#)

[FIGURE 3-12: Link styled to look like a button](#)

[FIGURE 3-13: WordPress-like options](#)

[FIGURE 3-14: Table style](#)

[FIGURE 3-15: Pagination style](#)

Chapter 4

[FIGURE 4-1: Insufficient privileges](#)

[FIGURE 4-2: Expired link message](#)

[FIGURE 4-3: Rogue JavaScript running](#)

[FIGURE 4-4: Related Posts list](#)

Chapter 7

[FIGURE 7-1: Classic Editor, not covered in this chapter](#)

[FIGURE 7-2: Gutenberg](#)

[FIGURE 7-3: Block Library menu](#)

[FIGURE 7-4: Categories of blocks](#)

[FIGURE 7-5: View options](#)

[FIGURE 7-6: Sidebar's Document menu](#)

[FIGURE 7-7: Sidebar's Block menu and formatting toolbar](#)

[FIGURE 7-8: WooCommerce blocks](#)

[FIGURE 7-9: Newest Products block](#)

[FIGURE 7-10: Event Calendar blocks](#)

[FIGURE 7-11: Post Type Switcher plugin](#)

[FIGURE 7-12: webpack finishing successfully.](#)

[FIGURE 7-13: Our “Hello world!” block in the Block Library](#)

[FIGURE 7-14: Selecting our new block](#)

[FIGURE 7-15: Editing a post](#)

[FIGURE 7-16: WP-CLI scaffold generated](#)

[FIGURE 7-17: Build Step 1](#)

[FIGURE 7-18: Build Step 2](#)

[FIGURE 7-19: Build Step 3](#)

[FIGURE 7-20: My Block in Block Library](#)

[FIGURE 7-21: My Block in Content Area](#)

Chapter 8

[FIGURE 8-1: Books admin menu and screen](#)

[FIGURE 8-2: Tags submenu item](#)

[FIGURE 8-3: Genres submenu](#)

Chapter 9

[FIGURE 9-1: New form on the user edit page](#)

[FIGURE 9-2: New roles](#)

Chapter 10

[FIGURE 10-1: Scheduled Events page](#)

[FIGURE 10-2: Number output](#)

Chapter 11

[FIGURE 11-1: Settings box](#)

Chapter 12

[FIGURE 12-1: Unformatted JSON](#)

[FIGURE 12-2: Formatted JSON](#)

[FIGURE 12-3: Post results](#)

[FIGURE 12-4: Error message](#)

[FIGURE 12-5: Authentication worked](#)

[FIGURE 12-6: “My Time at Crystal Lake” post](#)

Chapter 13

[FIGURE 13-1: Tools ⇔ Network menu options](#)

Chapter 14

[FIGURE 14-1: Simple post list](#)

[FIGURE 14-2: Widgets admin screen](#)

[FIGURE 14-3: Favorites list](#)

[FIGURE 14-4: Custom dashboard widget](#)

[FIGURE 14-5: Custom dashboard widget](#)

Chapter 15

[FIGURE 15-1: Admin Toolbar menu item](#)

[FIGURE 15-2: Query Monitor interface](#)

[FIGURE 15-3: Database queries](#)

Chapter 16

[FIGURE 16-1: Function with parameters](#)

[FIGURE 16-2: Codex search options](#)

[FIGURE 16-3: WordPress PHPXref](#)

[FIGURE 16-4: Function list](#)

PROFESSIONAL WORDPRESS® PLUGIN DEVELOPMENT

Second Edition

Brad Williams

Justin Tadlock

John James Jacoby



A Wiley Brand

FOREWORD

This book will teach you how to develop for WordPress. WordPress has, over the past two decades, grown into the CMS that powers more than one-third of all websites. If you're proficient at WordPress development, you'll never be out of a job again.

Starting out as a simple blogging system, over the last few years WordPress has morphed into a fully featured and widely used content management system. It offers individuals and companies worldwide a free and open source alternative to closed source and often very expensive systems.

When I say fully featured, that's really only true because of the ability to add any functionality needed in the form of a plugin. The core of WordPress is simple: you add in functionality with plugins as you need it. Developing plugins allows you to stand on the shoulders of a giant: you can showcase your specific area of expertise and help users benefit while not having to deal with parts of WordPress you don't care or know about.

When I wrote the foreword of this book's first edition, nine years ago, I'd just started my own company. That company has since grown to consist of 100+ people, and our plugins are used on more than 10 million sites—all through the power of open source and plugins.

I wished that when I started developing plugins for WordPress as a hobby, almost 15 years back, this book had been around. I used it as a reference countless times since, and I still regularly hand this book to new colleagues.

The authors of this book have always been a source of good information and wonderful forces in the WordPress community. Each of them is an expert in his own right; together they are one of the best teams that could have been gathered to write this book, and I'm glad they're here for a second edition.

WordPress makes it easy for people to have their say through words, sound, and visuals. For those who write code, WordPress allows you to express yourself in code. And it's simple. Anyone can write a WordPress plugin.

With this guide in hand, you can write a plugin that is true to WordPress' original vision: code is poetry.

Happy coding!

Joost de Valk

Yoast.com

INTRODUCTION

Dear reader, thank you for picking up this book! You have probably heard about WordPress already, the most popular self-hosted content management system (CMS) and blogging software in use today. WordPress powers literally millions of websites on the Internet, including high-profile sites such as TechCrunch and multiple Microsoft websites. What makes WordPress so popular is that it's free, open source, and extendable beyond limits. Thanks to a powerful, architecturally sound, and easy-to-use plugin system, you can customize how WordPress works and extend its functionalities. There are already more than 55,000 plugins freely available in the official plugin repository, but they won't suit all your needs or client requests. That's where this book comes in handy!

As of this writing, we (Brad, Justin, and John) have publicly released more than 100 plugins, which have been downloaded millions of times, and that's not counting private client work. This is a precious combined experience that we are going to leverage to teach you how to code your own plugins for WordPress by taking a hands-on approach with practical examples and real-life situations you will encounter with your clients.

The primary reason we wanted to write this book is to create a preeminent resource for WordPress plugin developers. When creating plugins for WordPress, it can be a challenge to find the resources needed in a single place. Many of the online tutorials and guides are outdated and recommend incorrect methods for plugin development. This book is one of the most extensive collections of plugin development information to date and should be considered required reading for anyone wanting to explore WordPress plugin development from the ground up.

WHO THIS BOOK IS FOR

This book is for professional web developers who want to make WordPress work exactly how they and their clients want. WordPress has already proven an exceptional platform for building any type of site from simple static pages to networks of full-featured communities. Learning how to code plugins will help you get the most out of WordPress and have a cost-effective approach to developing per-client features.

This book is also for the code freelancers who want to broaden their skill portfolio, understand the inner workings of WordPress functionality, and take on WordPress gigs. Since WordPress is the most popular software to code and power websites, it is crucial that you understand how things run under the hood and how you can make the engine work your way. Learning how to code plugins will be a priceless asset to add to your résumé and business card.

Finally, this book is for hobbyist PHP programmers who want to tinker with how their WordPress blog works, discover the infinite potential of lean and flexible source code, and learn how they can interact with the flow of events. The beauty of open source is that it's easy to learn from and easy to give back in turn. This book will help you take your first step into a community that will welcome your creativity and contribution.

Simply put, this book is for anyone who wants to extend the way WordPress works, whether it is for fun or profit.

WHAT YOU NEED TO USE THIS BOOK

This book assumes you already have a web server and WordPress running. For your convenience, it is preferred that your web server runs on your localhost, as it will be easier to modify plugin files as you read through the book, but an online server is also fine.

Code snippets written in PHP are the backbone of this book. You should be comfortable with reading and writing basic PHP code or referring to PHP's documentation to fill any gaps in knowledge about fundamental functions. Advanced PHP code tricks are explained, so you don't need to be a PHP expert.

You will need to have rudimentary HTML knowledge to fully understand all the code. A basic acquaintance with database and MySQL syntax will help with grasping advanced subjects. To make the most of the chapter dedicated to JavaScript and Ajax, comprehension of JavaScript code will be a plus.

WHAT THIS BOOK COVERS

As of this writing, WordPress 5.5 is around the corner, and this book has been developed alongside this version. Following the best coding practices outlined in this book and using built-in APIs are keys to future-proof code that will not be deprecated when a newer version of WordPress is released. We believe that every code snippet in this book will still be accurate and up-to-date for several years, just as several plugins we coded many years ago are still completely functional today.

HOW THIS BOOK IS STRUCTURED

This book is, to date, one of the most powerful and comprehensive resources you can find about WordPress plugins. Advanced areas of the many WordPress APIs are covered, such as the REST API, cron jobs, and custom post types. This book is divided into three major parts. Reading the first five chapters is required if you are taking your first steps in the wonders of WordPress plugins. [Chapters 6](#) through [9](#) will cover most common topics in coding plugins, and understanding them will be useful when reading subsequent chapters. The remaining chapters cover advanced APIs and functions, can be read in any order, and will sometimes refer to other chapters for details on a particular function.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

WARNING *Boxes with a warning label like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.*

NOTE *The note label indicates notes, tips, hints, tricks, and asides to the current discussion.*

As for styles in the text:

- We *italicize* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.

We present code in two different ways:

We use a monofont type with no highlighting for most code

examples.

We use **bold** to emphasize code that is particularly important in the present context or to show changes from a previous code snippet.

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All the source code used in this book is available for download at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

NOTE Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-1-119-66694-3.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time, you will be helping us provide even higher-quality information.

To find the errata page for this book, go to www.wiley.com and locate the title using the Search box. Then, on the book details page, click the Errata link. On this page, you can view all errata that have been submitted for this book and posted by editors. If you don't spot "your" error on the Errata page, go to support.wiley.com and follow the directions to contact technical support and open a ticket to submit the error. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent printings of the book.

1

An Introduction to Plugins

WHAT'S IN THIS CHAPTER?

- Understanding what a plugin is
- Using available WordPress APIs
- Finding examples of popular plugins
- Separating plugin and theme functionality
- Managing and installing plugins
- Understanding types of WordPress plugins

WordPress is the most popular open source content management system available today. One of the primary reasons WordPress is so popular is the ease with which you can customize and extend WordPress through plugins. WordPress has an amazing framework in place that gives plugin developers the tools needed to extend WordPress in any way imaginable.

Understanding how plugins work, and the tools available in WordPress, is critical knowledge when developing professional WordPress plugins.

WHAT IS A PLUGIN?

A *plugin* in WordPress is a PHP-based script that extends or alters the core functionality of WordPress. Quite simply, plugins are files installed in WordPress to add a feature, or set of features, to WordPress. Plugins can range in complexity from a simple social networking plugin to an extremely elaborate eCommerce package. There is no limit to what a plugin can do in WordPress; because of this, there is no shortage of plugins available for download.

How Plugins Interact with WordPress

WordPress features many different APIs for use in your plugin. Each API, or application programming interface, helps interact with WordPress in a different way. The following are the main available APIs in WordPress and their function:

- **Plugin:** Provides a set of hooks that enable plugins access to specific parts of WordPress. WordPress contains two different types of hooks: Actions and Filters. The Action hook enables you to trigger custom plugin code at specific points during execution. For example, you can trigger a custom function to run after a user registers a user account in WordPress. The Filter hook modifies text before adding it to or after retrieving it from the database.
- **Widgets:** Allows you to create and manage widgets in your plugin. Widgets appear under the Appearance ⇔ Widgets screen and are available to add to any registered sidebar in your theme. The API enables multiple instances of the same widget to be used throughout your sidebars.
- **Shortcode:** Adds shortcode support to your plugin. A shortcode is a simple hook that enables you to call a PHP function by adding something such as [shortcode] to a post or page.
- **HTTP:** Sends HTTP requests from your plugin. This API retrieves content from an external URL or for submitting content to a URL. Currently you have five different ways to send an HTTP request. This API standardizes that process and tests each method prior to executing. Based on your server configuration, the API will use the appropriate method and make the request.
- **REST API:** Allows developers to interact with your WordPress website remotely by sending and receiving JavaScript Object Notation (JSON) objects. You can create, read, update, and delete (CRUD) content within WordPress. The REST API is covered extensively in [Chapter 12](#), “REST API.”
- **Settings:** Inserts settings or a settings section for your plugin. The primary advantage to using the Settings API is security. All settings data is scrubbed, so you do not need to worry about cross-site request

forgery (CSRF) and cross-site scripting (XSS) attacks when saving plugin settings.

- **Options:** Stores and retrieves options in your plugin. This API features the capability to create new options, update existing options, delete options, and retrieve any option already defined.
- **Dashboard Widgets:** Creates Dashboard widgets. Widgets automatically appear on the WordPress Dashboard and contain all standard customization features including minimize, drag/drop, and screen options for hiding.
- **Rewrite:** Creates custom rewrite rules in your plugin. This API enables you to add static endpoints (/custom-page/), structure tags (%postname%), and feed links (/feed/json/).
- **Transients:** Creates temporary options (cached data) in your plugins. This API is similar to the Options API, but all options are saved with an expiration time.
- **Database:** Accesses the WordPress database. This includes creating, updating, deleting, and retrieving database records for use in your plugins.
- **Theme Customization (Customize) API:** Adds custom website and theme options to the WordPress Customizer. Theme customizations are displayed in a real-time preview prior to publishing to the live website.

There are additional, lesser known APIs that exist within the WordPress Core software. To view a full list, visit the Core Developer Handbook:

<https://make.wordpress.org/core/handbook/best-practices/core-apis>

WordPress also features pluggable functions. These functions enable you to override specific core functions in a plugin. For example, the `wp_mail()` function is a pluggable function. You can easily define this function in your plugin and send email using the Simple Mail Transfer Protocol (SMTP) rather than the default method. All pluggable functions are defined in the `/wp-includes/pluggable.php` WordPress Core file.

As an example, let's look at the `wp_mail()` pluggable function, which starts with this line of code:

```
if ( ! function_exists( 'wp_mail' ) ) :
```

You can see that the code first checks to see whether a `wp_mail()` function already exists using the `function_exists()` PHP function. If you created your own custom `wp_mail()` function, that will be used; if not, the WordPress Core version of `wp_mail()` will be used.

WARNING *Pluggable functions are no longer being added to WordPress Core. Newer functions utilize hooks for overriding their functionality.*

You can use some predefined functions during specific plugin tasks, such as when a plugin is activated or deactivated and even when a plugin is uninstalled. [Chapter 2](#), “Plugin Framework,” covers these functions in detail.

When Are Plugins Loaded?

Plugins are loaded early in the process when a WordPress-powered web page is called. [Figure 1-1](#) shows a high-level diagram of the standard loading process when loading a page in WordPress.

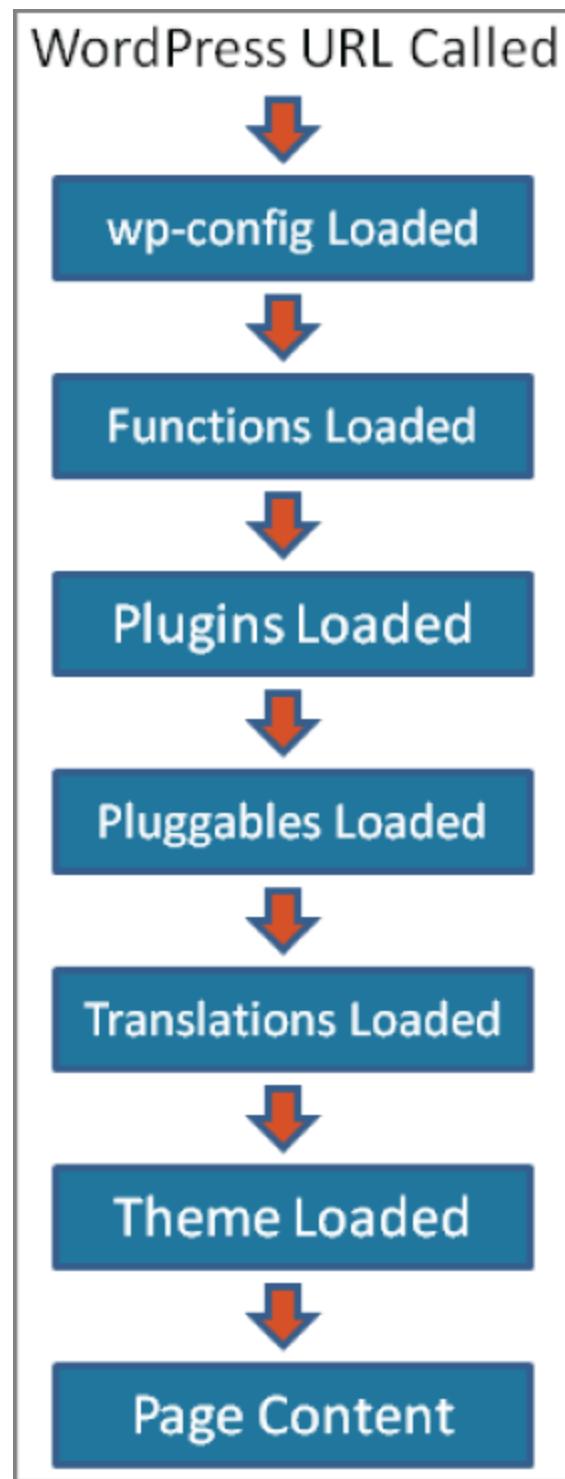


FIGURE 1-1: Loading a page in WordPress

The flow changes slightly when loading an admin page. The differences are minor and primarily concern what theme is loaded: admin theme versus your website theme.

AVAILABLE PLUGINS

When researching available plugins, you need to know where to find WordPress plugins. You can download plugins from many places on the Internet, but this isn't always a good idea.

WARNING *As with any software, downloading plugins from an untrusted source could lead to malware-injected and compromised plugin files. It's best to download plugins only from trusted websites and official sources such as the official Plugin Directory.*

Official Plugin Directory

The first place to start when researching available WordPress plugins is the official Plugin Directory at wordpress.org. The Plugin Directory is located at <https://wordpress.org/plugins>. With more than 55,000 plugins available and millions of plugin downloads, it's easy to see the vital role plugins play in every WordPress website. All plugins available in the Plugin Directory are 100 percent GPL and free to use for personal or commercial use.

Popular Plugin Examples

Take a look at some of the more popular WordPress plugins available to get a sense of their diversity:

- **Yoast SEO:** Advanced search engine optimization functionality for WordPress. Features include custom metadata for all content, canonical URLs, custom post type support, XML sitemaps, and more!
 - <https://wordpress.org/plugins/yoast-seo>
- **WPForms:** A powerful drag-and-drop form builder. Create simple contact forms and powerful subscription payment forms, all without writing a single line of code.
 - <https://wordpress.org/plugins/wpforms-lite>

- **BuddyPress:** A suite of components used to bring common social networking features to your website. Features for online communities include member profiles, activity streams, user groups, messaging, and more!
 - <https://wordpress.org/plugins/buddypress>
- **WooCommerce:** Advanced eCommerce solution built on WordPress. This is an extremely powerful plugin allowing anyone to sell physical and digital goods online.
 - <https://wordpress.org/plugins/woocommerce>
- **Custom Post Type UI:** Easy-to-use interface for registering and managing custom post types and taxonomies in WordPress.
 - <https://wordpress.org/plugins/custom-post-type-ui>

As you can see, the preceding plugins can handle a variety of complex tasks. The features added by these plugins are universal and features that many websites on the Internet could have.

Popular Plugin Tags

Now you will look at some popular tags for plugins. Plugin tags are just like blog post tags, simple keywords that describe a plugin in the Plugin Directory. This makes it easy to search for existing plugins by tag. The following are popular examples:

- **Twitter:** Everyone loves Twitter for micro-blogging and sharing links. You can find an abundance of Twitter-related plugins for WordPress.
 - <https://wordpress.org/plugins/tags/twitter>
- **Google:** With so many different services and APIs, *Google* is a popular plugin tag. Everything from Google ads to Google maps have been integrated into a WordPress plugin.
 - <https://wordpress.org/plugins/tags/google>
- **Blocks:** Most plugins that include block editor integration also use the *blocks* tag. This is great for viewing the many different types of blocks available for WordPress.

► <https://wordpress.org/plugins/tags/blocks>

Viewing popular plugin tags can provide inspiration when developing new plugins for WordPress.

ADVANTAGES OF PLUGINS

WordPress offers many advantages when using plugins. It's important to understand the advantages of building plugins to truly understand why you should spend time building them. This can also help when determining the need for a specific plugin in WordPress.

Not Modifying Core

One of the main advantages to plugins is the ability to modify the behavior of WordPress without modifying any core files. Core files refer to any file that is part of the default WordPress installation.

Hacking core files can make it difficult to update WordPress when a new version is released. If you made any modifications to a core file, that modification would be overwritten when the update occurs. Keeping WordPress up-to-date with the latest version is essential in keeping your website secure.

Modifying core files can also lead to an unstable website. Different areas of WordPress rely on other areas to function as expected. If you modify a core file and it no longer works as expected, it can cause instability and quite possibly break a completely unrelated feature in WordPress.

Why Reinvent the Wheel?

Another advantage to building plugins is the structure that already exists for your plugin. Many of the common features have already been developed and are ready for use in your plugin. For example, you can take advantage of the built-in user roles in WordPress. Using the user roles, you can easily restrict your code to execute only if a user is an administrator. Look at this example:

```
<?php
if ( current_user_can( 'manage_options' ) ) {
```

```
//any code entered here will only be executed IF
//user is an administrator
}
?>
```

As you can see, it's easy to verify that a user has proper permissions prior to executing any code in your plugin. You will learn about user accounts and roles in [Chapter 9](#), “Users and User Data.”

As another example, look at sending an email in WordPress. Sure, you could create a new function in your plugin to send email, but why? WordPress has a handy function called `wp_mail()` for sending email. Look at this example:

```
<?php
$email_to = 'you@example.com';
$email_subject = 'Plugin email example';
$email_message = 'How do you like my new plugin?';

wp_mail( $email_to, $email_subject, $email_message );
?>
```

As you can see, sending an email in WordPress couldn't be easier. Unless your plugin needs some customized emailing functionality, you don't need to re-create this function from scratch. Using this function also ensures the widest adoption for sending emails from WordPress because you use the built-in function.

Using the available built-in features of WordPress can greatly reduce the time to develop a plugin. Another advantage of not reinventing the wheel is that this approach more often than not will allow for your plugins to work across a greater number of servers and setups, thereby maximizing compatibility. Don't reinvent the wheel with features that already exist in WordPress.

Separating Plugins and Themes

A plugin can take control of the rendering process; therefore, the plugin can become a “theme.” Similarly, a theme can have plugin functionality included. Because of this, the difference between the two can sometimes become blurred, so why not just include your plugin code directly in a

theme? This is a common question and one that can have a few different answers.

Should themes include plugin functionality? The short answer is no. The primary reason for this is because plugins are meant to add features and functionality to WordPress, regardless of the theme used. This creates a nice separation between your website design and the functionality of your website. The reason this separation is needed is so your theme is not directly tied to the functionality required. WordPress is built so that you can easily change your design, or theme, at any point with just a couple clicks. If all plugin functionality existed in your theme and you switched themes, you will have lost all that functionality you required.

There is also a strong argument that certain features should be included in a theme. A common feature most themes include is breadcrumb navigation. This feature could certainly exist in a plugin, and there are many plugins available for this, but it makes more sense to include this navigation-centric feature in the theme.

Easy Updates

WordPress makes it easy to update a plugin to the latest version. Every plugin installed from the [WordPress.org](https://wordpress.org) Plugin Directory alerts you when a new version of the plugin has been released. Updating the plugin is as simple as clicking the update notification listed just below the plugin details on the Plugin screen of your WordPress Dashboard.

Plugins not installed from the Plugin Directory can also be updated using the auto-update functionality of WordPress. This is the method that premium plugins, specifically plugins that are sold on third-party websites outside of the Plugin Directory, push out updates to their plugins. The plugin author must define where WordPress can download the latest version, and WordPress will take care of the rest. If the plugin author doesn't define this location, you must manually update the plugin.

Keeping plugins updated is an important part of keeping your website free from security vulnerabilities and bugs.

Easier to Share and Reuse

Plugins are easy to share with others. It's much easier to share a plugin than tell someone to modify specific lines of code in your theme or WordPress. Using plugins also makes it easy to use the same functionality across multiple sites. If you find a group of plugins that you like, you can easily install them on every WordPress website you create.

Plugin Sandbox

When you activate a broken plugin in WordPress, it won't break your site. If the plugin triggers a fatal error, WordPress automatically deactivates the plugin before it has a chance to wreak havoc. This fail-safe feature makes it less risky when activating and testing new plugins. Even if the plugin does cause a white screen of death (error message), you can easily rename the plugin folder directly on your web server, and WordPress deactivates the plugin. This makes it impossible for a rogue plugin to lock you out of your own site because of an error.

On the other hand, if you were to hack the WordPress Core, you could cause fatal errors that would crash your website. This can also include causing unrecoverable damage to WordPress.

Plugin Community

A huge community is centered around plugin development, sharing knowledge and code, and creating amazing plugins. Getting involved in the community is a great way to take your plugin development skills to the next level. [Chapter 16](#), “The Developer Toolbox,” covers many of these resources.

INSTALLING AND MANAGING PLUGINS

All plugin management in WordPress happens on the Plugins screen in the WordPress Dashboard. The menu shown in [Figure 1-2](#) is available only to administrators in WordPress, so non-administrators cannot see this menu. If you use the Multisite feature of WordPress, the Plugins menu is hidden by default. You need to enable the menu using My Sites \Rightarrow Network Admin \Rightarrow Settings.

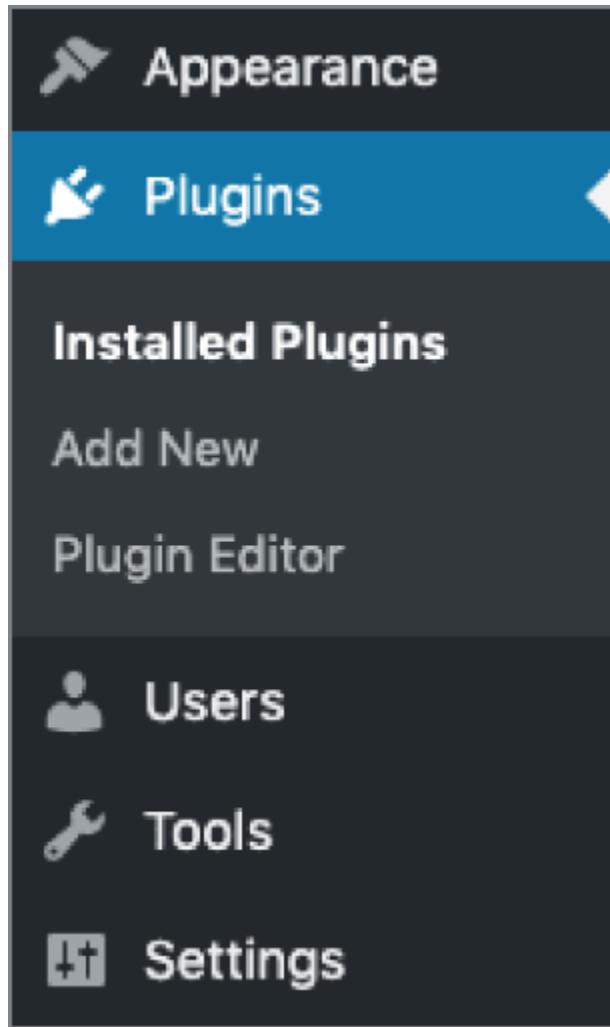


FIGURE 1-2: Plugins menu

Installing a Plugin

WordPress features three different methods for installing a new plugin. Your server setup dictates which method is the best to use.

The first method uses the built-in auto-installer. This method enables you to search the Plugin Directory on [WordPress.org](https://wordpress.org) directly from the Dashboard of your WordPress website. Simply visit Plugins ⇨ Add New from your WordPress Dashboard to search for a plugin. After you find a plugin to install, click the Install Now button, and the plugin automatically downloads and installs.

The second method uses the zip uploader. Zipped plugin files can be uploaded, extracted, and installed by WordPress. To use this method, click

the Upload Plugin button at the top of the Add Plugins page. Click the Choose File button and select the plugin zip file you want to install. After you select the plugin, click the Install Now button, as shown in [Figure 1-3](#).

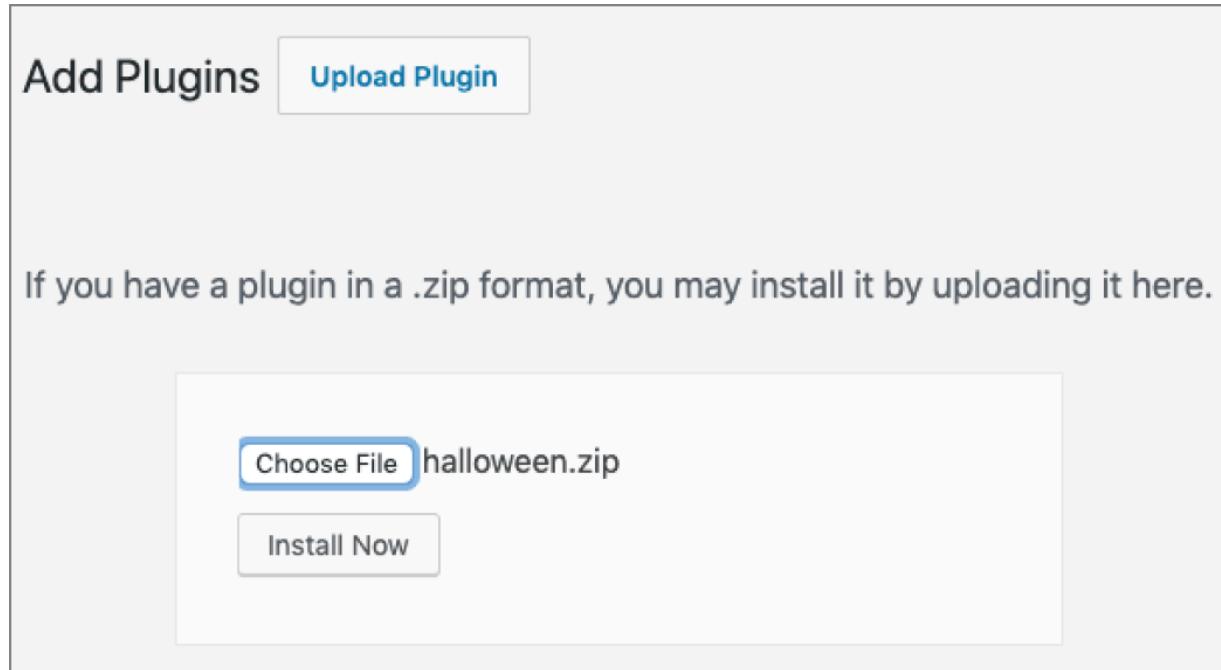


FIGURE 1-3: Install Now button

The third and final method to install a plugin in WordPress uses Secure (or SSH) File Transfer Protocol (SFTP). Using SFTP involves simply connecting to your web server using an SFTP client and manually uploading the plugin to your WordPress installation. To use this method, upload the uncompressed plugin folder or file to the `wp-content/plugins` directory on your web server.

Managing Plugins

After you install a plugin in WordPress, you can manage it, along with all other plugins, on the Plugins \Rightarrow Installed Plugins screen. Here you can find a list of all plugins, active or not, available in your WordPress installation. You can easily activate, deactivate, edit, update, and delete plugins from this screen.

The Plugin screen also features bulk actions for activating, deactivating, updating, and deleting plugins. Check all the plugins you want to manage

and then select the appropriate bulk action from the drop-down menu. This process makes managing multiple plugins a breeze!

Editing Plugins

WordPress features a built-in plugin editor on the Plugins \Rightarrow Plugin Editor screen. The plugin editor enables you to view and edit the source code of any plugin installed in WordPress. Keep in mind you can edit the source code only if the plugin file is writable by the web server; otherwise, you can only view the code.

To use the editor, select the plugin from the drop-down menu on the top-right portion of the Edit Plugins page. The editor lists all files associated with the selected plugin. There is also a documentation lookup feature that makes it easy to research a specific function's purpose in the plugin you are reviewing.

WARNING *A word of caution when using the built-in plugin editor: a browser doesn't have an Undo button. There is also no code revision history, so one bad code edit can crash your entire website with no way to revert the changes. It's best to use the code editor for reference only and never use it to edit your plugin files.*

Plugin Directories

A lesser known fact is that WordPress actually features two plugin directories. The primary directory is located at `wp-content/plugins` in a standard WordPress installation. The second, lesser known plugin directory is at `wp-content/mu-plugins`. The `mu-plugins` directory, which stands for *Must-Use*, is not automatically created by WordPress, so it must be manually created to be used.

The primary difference between the two is that the `mu-plugins` directory is for plugins that are always executed. This means any plugin included in this directory will automatically be loaded in WordPress and across all sites in the network if you run Multisite. Mu-plugins are always on and cannot be deactivated.

NOTE *The mu-plugins directory will not read plugins in a subfolder, so all plugins must be individual files or must include additional files that exist in a subdirectory. Any plugin files in a subfolder will be ignored unless included in the primary plugin file.*

Types of Plugins

WordPress features a few different types and statuses for plugins, as shown in [Figure 1-4](#). You need to understand the difference when administering and creating plugins for WordPress.



[FIGURE 1-4](#): Types and statuses for plugins

- **Active:** Plugin is active and running in WordPress.
- **Inactive:** Plugin is installed but not active. No code from the plugin is executed.
- **Recently Active:** A temporary status given to any plugin that has been recently deactivated.
- **Must-Use:** All plugins installed in the wp-content/mu-plugins directory. All Must-Use, or MU, plugins are loaded automatically. The only way to deactivate an MU plugin is to remove it completely from the directory.
- **Drop-ins:** Core functionality of WordPress can be replaced by Drop-in plugins. These plugins are specifically named PHP files located in the wp-content directory. If WordPress detects one of these files, it will be automatically loaded and listed under the Drop-in filter on the Plugin screen. Currently ten Drop-in plugins are available.
 - advanced-cache.php: Advanced caching plugin

- `db.php`: Custom database class
- `db-error.php`: Custom database error message
- `install.php`: Custom installation script
- `maintenance.php`: Custom maintenance message
- `object-cache.php`: External object cache
- `sunrise.php`: Advanced domain mapping
- `blog-deleted.php`: Custom blog deleted message
- `blog-inactive.php`: Custom blog inactive message
- `blog-suspended.php`: Custom blog suspended message

The last four Drop-in plugins are specific to the WordPress Multisite feature. A standard WordPress installation will have no use for these plugins.

When developing a new plugin, determine what type of plugin you want to create before you start the development process. Most plugins will be standard WordPress plugins, but occasionally you might need to create a Must-Use or Drop-in plugin.

SUMMARY

In this chapter, you learned about plugins and how they can interact with WordPress using the available APIs. The major advantages to using plugins and why plugin functionality shouldn't always be included in a theme were discussed. Installing and managing plugins in the WordPress Dashboard was covered.

Now that you understand how plugins work in WordPress, it's time to create the plugin foundation!

2

Plugin Framework

WHAT'S IN THIS CHAPTER?

- Creating a solid plugin foundation
- Determining directory and URL paths
- Creating activation and deactivation functions
- Cleaning up during the uninstall process
- Writing code following coding standards
- Properly documenting plugin code

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

When creating a plugin for WordPress, it's important to start with a solid foundation. This means getting your plugin folders and files organized and coming up with a naming scheme that you can use consistently throughout the plugin. The setup is the most essential piece of the puzzle. By nailing down these basics, you'll be well on your way to building your first WordPress plugin.

REQUIREMENTS FOR PLUGINS

The WordPress plugin system is flexible and allows for plugins to be either a single file or a folder with many files. In this section, you'll learn the basics of creating a plugin.

Naming Your Plugin

The most important thing when naming your plugin is for it to be something unique. It's also good practice for the name to reflect what the plugin actually does. For example, you wouldn't want to name a forum plugin "Joe's Plugin" because that doesn't tell potential users anything about what your plugin does.

You also need to consider whether your plugin name is too generic. It's possible that the name has already been taken by another plugin developer. You can check existing plugins in the WordPress Plugin Directory (<https://wordpress.org/plugins>).

Because there are thousands of existing plugins, some developers prefix their plugin name with their business name. For example, if your business name is "Super Duper," you might name a potential forum plugin "Super Duper Forums." This allows you to better attach your brand to your plugin and keep the name unique.

Using a Folder

While WordPress does allow plugins to be a single file, it is generally not good practice to use this method. Instead, it is standard practice to use a folder to contain your plugin. The vast majority of plugins will contain multiple files, such as PHP files, CSS and JavaScript assets, and other build files.

When creating a plugin, it should ideally match your plugin name. Plugin folder names should be hyphenated and lowercase. It should not contain spaces, underscores, uppercase letters, or nonalphanumeric characters. The plugin folder name must also match the plugin text domain when preparing your plugin for translations (see [Chapter 11](#), "Internationalization") if you plan to submit it to the official plugin repository.

Using the previous "Super Duper Forums" plugin as an example, its folder name should be `super-duper-forums`.

BEST PRACTICES

With any code base, developers should follow a common set of best practices. This is no different with WordPress plugins. By strictly following

the practices laid out in this section, you'll have a solid foundation for your plugin. You'll also learn how to organize your plugin files and subfolders.

Namespace Everything

If you have a function named `get_post()`, it will conflict with WordPress' own `get_post()` function and result in a fatal error. That's not a good user experience. Writing good code means making sure that your code doesn't conflict with other developers' code. The best way to ensure that is to prefix or namespace all your classes, functions, and anything else within the global namespace.

Throughout most of WordPress' existence, it has supported versions of PHP versions earlier than 5.3, which is the version of PHP that standardized namespaces. This meant that it was standard practice to use a *faux* namespace by prefixing classes and functions. Prefixes should be unique to the plugin. The WordPress coding standards recommend naming classes and functions using snake case. Therefore, a function named `get_post()` would become `prefix_get_post()`, and a class named `Post` would become `Prefix_Post`.

WordPress 5.2 changed its minimum PHP requirements to PHP 5.6. This means that plugin authors who want to support the same PHP versions as WordPress are no longer tied to the old practice of using prefixes. For this reason, it's probably best to look at other standards for naming, such as those outlined in the PHP-FIG standards (<https://www.php-fig.org>).

For the purposes of this book, the recommendation for namespacing your code will be to use the built-in method of namespacing in PHP. Namespaces allow you to group your code under a unique name so that it doesn't conflict with other developers' code. Using the “Super Duper Forums” plugin example, your namespace would be called `SuperDuperForums`.

Namespaces must be the first code, excluding the opening `<?php` tag and inline comments, in a PHP file. In the following example, see how a custom class named `Setup` is handled. Because it is under the `SuperDuperForums` namespace, there's no need to prefix it.

```
<?php
namespace SuperDuperForums;
```

```
class Setup {  
    public function boot() {}  
}
```

Class and function names are not the only consideration when namespacing code. There are other times when you'll have things in the global namespace that might conflict with other plugins running in WordPress. Such cases may include storing options in the database (see [Chapter 3](#), “Dashboard and Settings”) or loading a JavaScript file (see [Chapter 6](#), “JavaScript”). In such cases, it's generally best practice to prefix option names with your plugin name, such as `super_duper_forums_option_name`, and prefix script handles similarly, such as `super-duper-forums-script-name`.

You may have noticed that the option name used snake case (underscores) and the script handle used kebab case (hyphenated) in the previous example. The difference in practice has arisen over the years and can be confusing to new plugin developers. In general, anything referred to as a *handle* is in kebab case, and everything else is in snake case. Often such differences are superficial and not overly important. When in doubt, stick with snake case, and you'll be alright.

File Organization

When building your plugin, it's important to think about how your files should be organized. While you can change some things around later, getting started with a solid structure will make it easier to build out the plugin's features in the long run.

The most important and only required file for any plugin to have is its primary PHP file. This file can be named anything and must sit in the root of your plugin folder. If your primary file is named `plugin.php`, it will be located at `/super-duper-forums/plugin.php`.

There are two common practices for naming the primary plugin file. Many plugin authors name this file the same as the folder, such as `super-duper-forums.php`. The less-common practice is to simply name this `plugin.php`. You may decide to choose something else entirely. Over time as you develop plugins, you'll likely want to have a standard naming scheme that best suits you or your team.

In modern plugin development, you'll most likely have JavaScript build tools, config files, and all sorts of other files that take up space in the root of your plugin folder. Putting all your plugin files in the root will likely clutter things and make it hard to find things. It's best to only keep your primary plugin file in the root and all other plugin code within subfolders. The exception to this rule would be the `uninstall.php` file (see the “Uninstall.php” section later in this chapter).

Folder Structure

In professional development, it's important to create a folder structure that is understandable at a glance and easy to maintain. Most plugin developers will create separate folders to house their PHP code apart from resources, assets, or other front-end code in a folder with a name such as `/src`, `/inc`, `/includes`, or `/app`. Within the larger PHP development community, it is standard practice to name this folder `/src` because this is the folder where your “source” code lives.

The naming of the assets or resources folder is also varied with plugins. Often this folder is named `/assets`, `/dist`, or `/public`. Most likely, you'll want to have separate development and production folders if you use a build system such as `webpack` to bundle your final asset files.

The following is an example of how a plugin folder structure may be organized. Some of the files in this list are covered later in this chapter.

- `plugin.php`: Primary plugin file
- `uninstall.php`: Uninstall file
- `/src`: PHP source code
 - `Activator.php`: Activation class
 - `Deactivator.php`: Deactivation class
 - `Plugin.php`: Primary plugin class
- `/resources`: Development asset files
- `/css`: Development CSS
- `/js`: Development JavaScript

- /public: Production asset files
 - /css: Production CSS
 - /js: Production JavaScript

This is a clean structure that will help you maintain your plugin code over time. You may choose a different structure for your own projects.

NOTE *The most important thing is consistency. In a professional environment, more than one developer will often be working on code, so it's important that your team understands where to find and edit code.*

When using namespaces as described in the “Namespace Everything” section of this chapter, it is standard practice to have a folder structure that matches how your namespaces and subnamespaces are set up so that it is easy to autoload the files using a system that follows the PHP-FIG autoloading standard (<https://www.php-fig.org/psr/psr-4>).

With a fully qualified class name of PDEV\Post\Edit, you'd have the following structure within your /src folder:

- /Post: Subnamespace
 - Edit.php: Class file

Following this structure will tie your code and namespace structure directly to your folder structure. It makes it simple for you and other developers to navigate your project's files and folders by just looking at the namespace and class name.

This is the standard used by the larger PHP community and has been for many years. Much of the WordPress community is still light-years behind in following this standard, but it is beginning to catch on with more and more plugin developers.

PLUGIN HEADER

For WordPress to recognize a plugin, the plugin's primary PHP file must have what's called a *plugin header* at the top of the file. This tells WordPress that this particular file is the file that it must load to your plugin. WordPress plugins can be and do anything, but this is the only hard requirement for a plugin to function.

Creating the Header

The plugin header is nothing more than a normal PHP inline code comment with some special formatting that WordPress can recognize. The following is an example of a plugin header:

```
<?php
/**
 * Plugin Name: My Plugin
 * Plugin URI: https://example.com/plugins/pdev
 * Description: A short description of the plugin.
 * Version: 1.0.0
 * Requires at least: 5.3
 * Requires PHP: 5.6
 * Author: John Doe
 * Author URI: https://example.com
 * License: GPL v2 or later
 * License URI: https://www.gnu.org/licenses/gpl-2.0.html
 * Text Domain: pdev
 * Domain Path: /public/lang
 */
```

Each line is called a *header file* and provides some info to WordPress about your plugin. The only required field for the plugin to work is the `Plugin Name` field. Each field should be straightforward to understand. The following is a brief description of what each field sets:

- `Plugin Name`: The name for your plugin
- `Plugin URI`: A URI with more information about your plugin
- `Description`: A brief summary that describes what your plugin does
- `Version`: The current version of your plugin, which should be incremented with each update

- Requires at least: The minimum version of WordPress required for your plugin to work
- Requires PHP: The minimum version of PHP required for your plugin to work
- Author: Your or your team/business name
- Author URI: The link to your website
- License: The license that the plugin is distributed under
- License URI: A link to the full text of the license
- Text Domain: The text domain used for internationalizing your plugin (see [Chapter 11](#) for more information)
- Domain Path: The relative path to where translation files are located in your plugin
- Network: An optional field that can be set to true for plugins that can only be activated across the entire network on a multisite installation

Plugin License

When distributing your plugin to others, it's important to have a clear license so that you both protect your own copyright and make sure anyone who receives your plugin has a clear understanding of what their rights are. WordPress is licensed under the GNU General Public License (GPL), version 2 or later. Any plugins distributed should use a license that is compatible with the GPL.

The GPL recommends placing the following copyright notice in every source file for your plugin. However, most plugins only place this after the plugin header in the primary plugin file.

```
<?php
/*
Copyright (C) <year> <name of author>
```

```
This program 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
(at your option) any later version.

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 Street, Fifth Floor, Boston, MA 02110-1301 USA.
*/

You need to replace the <year> and <name of author> snippets in that
notice with the year the code was created and your name. Once you add the
notification to your plugin file(s), it will then be licensed under the GPL.

If working with a client or employer and not distributing the plugin to the
public in any way, your plugin may not need a license. Copyright and/or
license should be determined by your contract and what you and your
client/employer agree to.

DETERMINING PATHS

Perhaps one of the toughest hurdles to jump when building WordPress
plugins is simply figuring out the appropriate path when loading or
referencing a file. Because you as a plugin developer do not control how the
user chooses to set up their WordPress installation, you cannot rely on a
specific pattern.

Plugin Paths

There are two types of paths that you might be concerned with when
building WordPress plugins. The first is the local path to files on the server.
The second is URL paths, which may be necessary for loading JavaScript
files, CSS stylesheets, or other assets. You may also need to link to a
specific URL within the WordPress installation itself.

Because WordPress provides the ability for users to move their `wp-content` folder (where plugins are located) to any location on the server, it's important to use the standard WordPress functions for determining the correct path.

Local Paths

Local paths reference locations on the server. PHP provides easy methods for determining paths for almost every case needed. However, to be on the safe side, it's best to use WordPress functions where possible. The `plugin_dir_path()` function provides an easy way to get the filesystem directory path to your plugin.

```
<?php $path = plugin_dir_path( $file ); ?>
```

Parameters:

► `$file` (string, required): The filename in the current directory path

The following example will print the path of the current directory of the file passed in with a trailing slash using PHP's `__FILE__` constant:

```
<?php echo plugin_dir_path( __FILE__ ); ?>
```

That code should output a path similar to the following:

```
/public_html/wp-content/plugins/pdev/
```

Any file can be passed into the function. The resulting path will point to wherever the file lives in your plugin. If passing in `__FILE__` from the primary plugin file, it'll be the path to the root of your plugin. If passing it in from a subfolder in your plugin, it will return the path to the subfolder in your plugin.

Assuming you needed to load a file named `/src/functions.php` from your plugin that houses some custom functions, use the following code:

```
<?php include plugin_dir_path( __FILE__ ) .  
  '/src/functions.php'; ?>
```

It is common practice for plugin authors to store this path as a variable or constant in the plugin's primary file for quick access to the plugin's root

folder path, as shown in the following example code:

```
<?php define( 'PDEV_DIR', plugin_dir_path( __FILE__ ) ); ?>
```

This allows you to reference `PDEV_DIR` any time you need it from anywhere in the plugin without having to think about file paths.

URL Paths

Referencing URL paths can be tougher than local paths because there's usually no good way to determine this via standard PHP functions or constants alone. You'll need to rely on WordPress to get the correct path.

The primary use case for getting a URL path will be determining the path to an asset (e.g., JavaScript, CSS, or image files) within your plugin.

WordPress provides the `plugin_dir_url()` function to handle this use case.

```
<?php $url = plugin_dir_url( $file ); ?>
```

Parameters:

➤ `$file` (string, required): The filename in the current directory path

WordPress will automatically convert any filename passed in to an appropriate URL equivalent. See the following example of passing in the filename from within the primary plugin file:

```
<?php echo plugin_dir_url( __FILE__ ); ?>
```

That code will output something like the following URL with a trailing slash:

`https://example.com/wp-content/plugins/pdev/`

If you wanted to determine the path of a JavaScript file located at `/public/js/example.js` in your plugin, you'd use the following code:

```
<?php $url = plugin_dir_url( __FILE__ ) .  
'public/js/example.js'; ?>
```

You can use this to correctly get the URL path to any location in your plugin. Like its `plugin_dir_path()` counterpart, you can pass in any filename in your plugin to determine the appropriate URL for any location.

WordPress also provides a `plugins_url()` function. `plugin_dir_url()` is a wrapper for this function. For most purposes, these two functions can almost be used interchangeably, but there are some important differences.

```
<?php $url = plugins_url( $path = '', $plugin = '' ); ?>
```

Parameters:

- `$path` (string, optional): A path to append to the end of the URL
- `$plugin` (string, optional): The full path to a file within the plugin

If no parameters are provided, the function will return the URL to the plugin's directory for the WordPress installation. It also does not add a trailing slash to the end of the URL. For most cases, it's usually best to stick with `plugin_dir_url()`. However, this function is available if needed.

Other than determining the URL path to files within your plugin, you may need to determine the URL for a particular page or directory within the WordPress installation. WordPress has a number of useful functions that return the necessary information.

- `site_url()`: URL path to where WordPress is installed
- `home_url()`: URL path to the site's homepage
- `admin_url()`: URL path to the WordPress admin
- `rest_url()`: URL path the REST API endpoint
- `includes_url()`: URL path to the WordPress includes directory
- `content_url()`: URL path to the WordPress content directory

All of these URL functions accept an optional first parameter of `$path`, which is appended to the end of the URL if provided. The following example shows how to retrieve the URL path to the General Settings page in the admin:

```
<?php $url = admin_url( 'options-general.php' ); ?>
```

With the exception of the `content_url()` function, each of these functions also accepts an optional second parameter of `$scheme`, which allows you to

manually set the protocol, such as 'http' or 'https'. In almost all cases, you should not set this parameter and should instead allow WordPress to automatically determine the appropriate URL scheme.

First-time WordPress developers will also sometimes confuse `site_url()` and `home_url()`. WordPress can be installed in a subdirectory on the server while allowing the actual site to be located elsewhere. Unfortunately, the function names are part of a legacy code base and have stuck around. The trick is to remember that the “home” in `home_url()` refers to the URL of the site, and the “site” in `site_url()` refers to the URL of the WordPress installation.

If WordPress is installed in a subdirectory, `site_url()` might point to a URL like <https://example.com/wordpress>, while `home_url()` points to <https://example.com>.

ACTIVATE/DEACTIVATE FUNCTIONS

WordPress provides standard activation and deactivation hooks that allow any plugin to run code when it is activated or deactivated, respectively. This section walks through how to use both.

Plugin Activation Function

When building plugins, you'll often need to execute some code when the user first activates it. One common use case is adding custom capabilities to the administrator role (see [Chapter 9](#), “Users and User Data”) for something like a custom post type (see [Chapter 8](#), “Content”) that your plugin is registering. Or, you may need to set up a particular option for your plugin to perform.

WordPress provides the `register_activation_hook()` function that allows you to pass in a callback for handling any activation code you need to run.

```
<?php register_activation_hook( string $file, callable  
$function ); ?>
```

Parameters:

- `$file` (string, required): Filesystem path to the primary plugin file

- `$function (callable, required):` The PHP callable to be executed when the plugin is activated

It's generally best to separate your activation code from the rest of your plugin code simply for organizational purposes. The following example shows how to register an activation hook that points to a class located at `src/Activation.php` in your plugin and executes its `activate()` method:

```
<?php

namespace PDEV;

register_activation_hook( __FILE__, function() {
    require_once plugin_dir_path( __FILE__ ) .
    'src/Activation.php';
    Activation::activate();
} );
```

With the activation hook callback registered, you need to figure out what things you need to set upon activation. The following example gets the administrator role object and adds a custom capability to it:

```
<?php

namespace PDEV;

class Activation {

    public static function activate() {

        $role = get_role( 'administrator' );

        if ( ! empty( $role ) ) {
            $role->add_cap( 'pdev_manage' );
        }
    }
}
```

There is no limit to what code you can run upon activation. It's a great time to run anything that needs to be set only once or things that shouldn't be run on every page load. For example, if your plugin needs to create custom database tables, this would be the time to do it.

While it is common practice with some plugin authors, you usually shouldn't set plugin database options at this point. It's almost always better to use sane defaults for any options your plugin has and not clutter the database until the end user has explicitly decided to save on your plugin settings screen (see [Chapter 3](#)).

Plugin Deactivation Function

Like the activation function, WordPress also allows you to execute code from a registered deactivation callback via the `register_deactivation_hook()` function. The following example includes the `src/Deactivation.php` class and executes its `deactivate()` method:

```
<?php

namespace PDEV;

register_deactivation_hook( __FILE__, function() {
    require_once plugin_dir_path( __FILE__ ) .
    'src/Deactivation.php';
    Deactivation::deactivate();
} );
```

Once you've registered your deactivation callback, you need to run some code on deactivation. The following is an example of how to set up your `Deactivation` class:

```
<?php

namespace PDEV;

class Deactivation {

    public static function deactivate() {
        // Run your deactivation code here.
    }
}
```

Like with the activation hook, you have the freedom to run any code that you need when a user decides to deactivate your plugin.

Deactivate Is Not Uninstall

Setting up a deactivation callback is often unnecessary because most of the things you might think you'd need to unset on deactivation should actually happen when a user uninstalls the plugin. Deactivation is not an appropriate time to delete plugin options, remove database tables, delete user content, or perform any other highly destructive actions.

Users and even WordPress itself may deactivate a plugin for any number of reasons. You don't want your plugin users to lose any important data when deactivating only to reactivate and find their settings or content gone. Use with extreme caution.

UNINSTALL METHODS

Cleaning up anything left behind when a user uninstalls your plugin is an important aspect of developing plugins professionally. Running an uninstall procedure takes little work on your part but makes sure that users can trust that they can use your plugins without those plugins leaving unwanted data behind.

Why Uninstall Is Necessary

If you install an app on your mobile phone and decide at some point that you no longer want to use that app, you'll delete or “uninstall” it. You wouldn't want the app to leave behind all of its settings or any other data that is no longer useful. WordPress plugins are no different. Users should feel confident that their site doesn't have cruft left over from every plugin they've used or even just tested and decide it wasn't for them.

If there's an ultimate “no-no” in plugin development, it's to not tidy up after yourself. When your plugin is uninstalled, make sure to delete any options or other data that your plugin has added.

However, there are times when you want to ignore this rule. In general, user-created content, such as that created by custom post types or custom database tables, may need to be retained. In these situations, it's best to provide a setting that a user can opt into that allows you to delete that data on uninstall. The rule of thumb here is to always be respectful of their content. When in doubt, get the user's permission before deleting anything they might want to keep.

WordPress provides two different methods for uninstalling a plugin: the `uninstall.php` file and the `uninstall hook`.

Uninstall.php

The first method of uninstalling a plugin is by creating an `uninstall.php` file in the root of your plugin folder. This is the method that you'll most likely be using and is the recommended route to take.

Using `uninstall.php` is the preferred method of uninstalling a plugin because it isolates your uninstall code from the rest of your plugin and doesn't allow arbitrary code to run from your other plugin files. Ideally, your plugin should be structured such that arbitrary code is never run. However, `uninstall.php` protects against this and makes sure only the code contained within the file is executed.

It's also important to check that `WP_UNINSTALL_PLUGIN` is set before executing your uninstall code. Otherwise, it's possible the file will be inadvertently called and your plugin will delete everything when it shouldn't.

In the following example, you can see that the uninstall procedure checks the constant and removes the `pdev_manage` capability added in the “Plugin Activation Hook” section earlier in this chapter:

```
<?php

if ( ! defined( 'WP_UNINSTALL_PLUGIN' ) ) {
    wp_die( sprintf(
        __( '%s should only be called when uninstalling
the plugin.', 'pdev' ),
        __FILE__
    ) );
    exit;
}

$role = get_role( 'administrator' );

if ( ! empty( $role ) ) {
    $role->remove_cap( 'pdev_manage' );
}
```

Uninstall Hook

WordPress provides an uninstall hook similar to its activation and deactivation hooks outlined earlier in this chapter. If no `uninstall.php` file exists for the plugin, WordPress will look for a callback registered via the `register_uninstall_hook()` function.

```
<?php register_uninstall_hook( $file, $callback ); ?>
```

Parameters:

- `$file` (string, required): The primary plugin file
- `$callback` (callable, required): A function to execute during the uninstall process

Now look at the following example of how a plugin uninstall function works:

```
<?php

register_uninstall_hook( __FILE__, 'pdev_uninstall' );

function pdev_uninstall() {
    $role = get_role( 'administrator' );

    if ( ! empty( $role ) ) {
        $role->remove_cap( 'pdev_manage' );
    }
}
```

That example performs the same action as the `uninstall.php` file from the previous section in this chapter. However, you may have noted an important difference. There's no need to check the `WP_UNINSTALL_PLUGIN` constant because it's not set in the uninstall callback and unnecessary.

It's also important to use `$this` or an object reference when registering an uninstall hook because this callback reference is stored in the database and is unique to the page load when it is stored.

CODING STANDARDS

Following a set of standards when developing plugins professionally is important because it makes long-term maintenance of your code easier and also allows other developers to quickly learn from or contribute to your plugin. WordPress maintains a coding standard for the core source code. It is a good starting point when building plugins. You can view the WordPress coding standards at <https://make.wordpress.org/core/handbook/best-practices/coding-standards>.

WordPress has more than 15 years of legacy code, and sometimes its standards do not align with the standards of the larger PHP community. This can be off-putting for some PHP developers new to the WordPress community. You may also want to follow PHP Recommended Standards (PSRs) created and maintained by the PHP Framework Interop Group (PHP-FIG) for the larger PHP community. You can find these standards at <https://www.php-fig.org/psr>.

The most important thing is to maintain consistency in your code base, regardless of the coding standards that you follow.

Document Your Code

One of the worst things you can do as a developer is to never document your code. While you should always strive for your code to be self-documenting by naming things in ways that make sense and limiting functions/classes to a single responsibility, documenting your code is just as important as the code itself.

Imagine writing a large plugin for a client today and not documenting what everything does. That same client calls you up in two years to make some big updates for them. Can you honestly say that you'll completely understand the code that you wrote two years ago? Even if you could, what about another developer? You may think that it's not your problem, but that could earn you a poor reputation in the WordPress developer community. You should be a "good citizen" and help your fellow developers.

Documentation is also important when building plugins for public release. For others to contribute code to your plugin through patches or pull requests, other developers should have enough documentation to understand any decisions that you made with the code.

WordPress uses PHPDoc for adding documentation to code. PHPDoc is a standard for documenting files, functions, classes, and any other code written in PHP. The following is an example of using PHPDoc to document a function:

```
<?php
/**
 * Short Description.
 *
 * Longer and more detailed description.
 *
 * @param type $param_a Description.
 * @param type $param_b Description.
 * @return type Description.
 */
function pdev_function( $param_a, $param_b ) {
    // Do stuff.
}
```

As you can see, the inline PHPDoc describes what the function does, what parameters it accepts, and what it returns. Another developer wouldn't even have to read your code to understand what it does.

NOTE *Good documentation helps when spotting bugs. If the function's code does something different from what is outlined by the documentation, you or another developer will know either the documentation or the code is incorrect.*

Naming Variables and Functions

Variables and functions should always be written in snake case. All characters should be written in lowercase, and multiple words should be separated with an underscore. Functions and variables in the global namespace should also be preceded with a unique prefix for your plugin, such as pdev_. The following example is the correct way to write a function:

```
<?php
function pdev_function( $param_a ) {
    // Do stuff.
}
```

`$param_a` isn't in the global scope there, so it doesn't need a prefix. While it's considered bad practice to use global variables, if you do use them, make sure to prefix.

Naming Classes and Methods

Class and method naming is one area where the core WordPress coding standards don't align with the PHP community. The PHP standard is to write class names in PascalCase where the first character of each word is uppercase and there are no separators between words. The standard for class methods is to write words in camelCase where the difference is that the first character of the first word is lowercase.

The following example shows what a basic class with a single method looks like using this method:

```
<?php  
  
class PDEVSetup {  
  
    public function setupActions() {  
    }  
}
```

The WordPress coding standards follow a different set of rules. Look at the following example to spot the differences:

```
<?php  
  
class PDEV_Setup {  
  
    public function setup_actions() {  
    }  
}
```

As you can see, the WordPress method uses underscores to separate class names and uses snake case for method names. You'll need to decide the best direction for your own plugins. There's no one right answer, and developers have been arguing over naming schemes for as long as programming has existed. The most important thing is to be consistent in your own code.

Naming Files

WordPress has a strict guideline on how it names files. The standard is to write each character in lowercase and separate multiple words with hyphens. For example, you might have a group of user-related functions in your plugin. You'd want to name that file something like `functions-users.php`.

WordPress also prefixes class filenames with `class-`. The `PDEVSetup` or `PDEV_Setup` class from the previous section in this chapter would be named `class-pdev-setup.php` following the WordPress standard.

Again, this is where WordPress deviates from the standards within the PHP community. At some point, you'll likely want to use an autoloader, such as the one provided by Composer (<https://getcomposer.org>). Once you do that, you'll quickly realize that the best method of naming class files is for the filename to match the class name exactly because it simplifies the autoloader code and keeps you from having to write a custom implementation. In this case, you should name the file for the `PDEVSetup` class as `PDEVSetup.php`.

Single and Double Quotes

PHP allows you to use single or double quotes when defining strings. With WordPress, it's recommended to always use single quotes when possible. This makes it easy to use double quotes within the string without escaping them. The following is an example of echoing a link that has double quotes within the single-quoted string:

```
<?php
echo '<a href="https://example.com">Visit Example.com</a>';
```

You can also use double quotes when you need to insert a variable within the string. The following example grabs the current site's name and echoes a message with the variable:

```
<?php
$pdev_site_name = get_bloginfo( 'name', 'display' );
echo "You are viewing {$pdev_site_name}.";
```

Indentation

Put 10 programmers in a room and pose the question, “Spaces or tabs?” This may be the closest you’ll ever see to a programming discussion resulting in fistcuffs. Many a developer has lived and died on their personal space/tab hill. The one thing that most can agree on is that your code should follow some sort of standard indentation.

The WordPress standard is to indent all code with tabs. Consider the following example of an if/else statement:

```
<?php
if ( $condition ) {
echo 'Yes';
} else {
echo 'No';
}
```

It's hard to follow the logical structure of the code with no indentation. Instead, you should indent each line within the brackets of the conditions, as shown in the following example:

```
<?php
if ( $condition ) {
    echo 'Yes';
} else {
    echo 'No';
}
```

The one exception that WordPress makes to its tab preference is when aligning multiple lines of similar items. Take note of the spacing usage in the following snippet:

```
<?php
$some_var      = 'ABC';
$another_var   = 'JKL';
$yet_another_var = 'XYZ';
```

As you can see, the operators and values assigned to the variables are clearly lined up and easy to read.

Brace Style

Any multiline statement in PHP should use brackets to contain the entire code block. Following this brace style will keep your code clear and less

prone to errors. See the next code snippet for the appropriate usage of braces:

```
<?php
if ( $condition_a ) {
    action_a();
} elseif ( $condition_b || $condition_c ) {
    action_b();
    action_c();
}
```

Even if there's only a single line within each block, make sure to enclose it too. Forgoing the braces for single-line statements is usually a bad idea and may result in bugs when you add extra code in the future.

Space Usage

When building your plugin, you should always add spaces after commas and on each side of logical, comparison, and other operators. The following code snippet shows several basic PHP examples and how they should be spaced:

```
<?php
if ( $foo === 'bar' ) {
    // Do something.
} elseif ( ! $foo ) {
    // Do something else.
}

foreach ( $foo as $bar ) {
    // Do something.
}

function pdev_function( $param_a = 'foo', $param_b = 'bar' )
{
    // Do something.
}

$foo = range( 10, 100, 10 );
```

Using this spacing technique keeps your code clean and makes it easy to read. The rule of thumb is to always be judicious with spacing. Or, when in doubt, use a space.

Shorthand PHP

You should never use shorthand PHP tags (`<?` and `?>`) when developing plugins. Shorthand tags must be enabled on the server to work. As a plugin developer, you typically won't have access to the server, so there's no guarantee they'll work for your users. Always use the full PHP opening and closing tags (`<?php` and `?>`).

The one exception to this rule is for the short echo tag, as shown in the following example:

```
<?= 'Hello, World!'; ?>
```

The short echo tag is always enabled as of PHP 5.4. This is uncommon usage and not standard for WordPress. It's typically used in templates, which is not the primary use case for plugins.

SQL Statements

When writing SQL statements in WordPress to make a direct database call, always capitalize the SQL part of the statement. Most statements should be written on a single line, but it's OK to break the statement into multiple lines if the statement is complex.

```
SELECT ID FROM wp_users WHERE user_login = 'example'
```

[Chapter 4](#), “Security and Performance,” covers how to create proper SQL statements in more detail.

SUMMARY

This chapter covered how to properly set up your WordPress plugin. Following the methods and techniques outlined in this chapter will set you on the path to building plugins professionally. Starting from a solid foundation by following coding standards and documenting your code will make it easier to maintain your plugin in the long term. It will also help other developers understand your code. It's important to get the foundation right before moving on to more advanced topics.

3

Dashboard and Settings

WHAT'S IN THIS CHAPTER?

- ▶ Creating menus and submenus
- ▶ Saving and retrieving plugin settings
- ▶ Saving global and per-user options
- ▶ Designing and styling your plugin

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

Integrating your plugin in WordPress is a critical step in building a professional plugin. WordPress features many different ways to integrate your plugin including adding top-level and submenu items, creating custom setting pages, and styling to match the existing Dashboard user interface (UI).

In this chapter, you learn how to properly integrate your plugin into the various areas of WordPress. You'll learn how to create a plugin settings pages to save and retrieve data using internal WordPress functions and API. You'll also learn the proper design and styles available that your plugins can take advantage of to provide your users with a consistent UI experience.

ADDING MENUS AND SUBMENUS

Many plugins you create need some type of menu item, which generally links to your plugin's settings page where users can configure your plugin options. WordPress features two methods for adding a plugin menu: a top-level menu or a submenu item.

Creating a Top-Level Menu

The first menu method for your plugin to explore in WordPress is a new top-level menu, which is added to the Dashboard menu list. For example, Settings is a top-level menu. A top-level menu is common practice for any plugin that needs multiple option pages. To register a top-level menu, you use the `add_menu_page()` function.

```
<?php add_menu_page( page_title, menu_title, capability,
menu_slug, function,
icon_url, position ); ?>
```

The `add_menu_page()` function accepts the following parameters:

- `page_title`: Title of the page as shown in the `<title>` tags.
- `menu_title`: Name of your menu displayed on the Dashboard.
- `capability`: Minimum capability required to view the menu.
- `menu_slug`: Slug name to refer to the menu; should be a unique name.
- `function`: Function to be called to display the page content for the item.
- `icon_url`: URL to a custom image to use as the menu icon. Also supports the `dashicons` helper class to use a font icon (e.g. `dashicons-chart-pie`).
- `position`: Location in the menu order where it should appear.

Now create a new menu for your plugin to see the menu process in action. Use the `admin_menu` action hook to trigger your menu code. This is the appropriate hook to use whenever you create menus and submenus in your plugins.

```
<?php
add_action( 'admin_menu', 'pdev_create_menu' );

function pdev_create_menu() {

    //create custom top-level menu
    add_menu_page( 'PDEV Settings Page', 'PDEV Settings',
        'manage_options', 'pdev-options',
        'pdev_settings_page',
```

```
        'dashicons-smiley', 99 );  
    }  
?>
```

As you can see, the `admin_menu` action hook calls your custom `pdev_create_menu()` function. Next you need to call the `add_menu_page()` function to register the custom menu in WordPress. Set the name of your menu to PDEV Settings, require that the user has `manage_options` capabilities (that is, is an administrator), and set the callback function to `pdev_settings_page()`. You also set the menu icon to use the built-in smiley dashicon (covered in detail later in this chapter). The final parameter is the menu position, which defines where the menu will appear within the WordPress Dashboard's left menu.

The result is a custom registered menu as shown in [Figure 3-1](#).



Home

Updates

Posts

Media

Pages

Comments

Appearance

Plugins

Users

Tools

Settings

PDEV Settings

FIGURE 3-1: Custom registered menu

NOTE *Menus are a common feature in WordPress plugins and are generally expected by users. It's a good idea to mention where your plugin settings can be found in the plugin description and documentation.*

Adding a Submenu

Now that you have a new top-level menu created, create some submenus for it, which are menu items listed below your top-level menu. For example, Settings is a top-level menu, whereas General, listed below Settings, is a submenu of the Settings menu. To register a submenu, use the `add_submenu_page()` function.

```
<?php add_submenu_page( parent_slug, page_title, menu_title,  
capability,  
menu_slug, function ); ?>
```

The `add_submenu_page()` function accepts the following parameters:

- `parent_slug`: Slug name for the parent menu (`menu_slug` previously defined)
- `page_title`: Title of the page as shown in the `<title>` tags
- `menu_title`: Name of your submenu displayed on the Dashboard
- `capability`: Minimum capability required to view the submenu
- `menu_slug`: Slug name to refer to the submenu; should be a unique name
- `function`: Function to be called to display the page content for the item

Now that you know how submenus are defined, you can add one to your custom top-level menu.

```
<?php  
add_action( 'admin_menu', 'pdev_create_menu' );
```

```
function pdev_create_menu() {  
    //create custom top-level menu  
    add_menu_page( 'PDEV Settings Page', 'PDEV Settings',  
        'manage_options', 'pdev-options',  
        'pdev_settings_page',  
        'dashicons-smiley', 99 );  
  
    //create submenu items  
    add_submenu_page( 'pdev-options', 'About The PDEV  
Plugin', 'About',  
        'manage_options', 'pdev-about', 'pdev_about_page' );  
    add_submenu_page( 'pdev-options', 'Help With The PDEV  
Plugin',  
        'Help', 'manage_options', 'pdev-help',  
        'pdev_help_page' );  
    add_submenu_page( 'pdev-options', 'Uninstall The PDEV  
Plugin',  
        'Uninstall', 'manage_options', 'pdev-uninstall',  
        'pdev_uninstall_page' );  
}  
?>
```

The preceding code creates three submenus for your custom top-level menu: About, Help, and Uninstall, as shown in [Figure 3-2](#). Each of these submenu items links to a different custom function that can contain any code you want to use for that submenu page.

 Dashboard

 Posts

 Media

 Pages

 Comments

 Appearance

 Plugins

 Users

 Tools

 Settings

 PDEV Settings <

PDEV Settings

About

Help

Uninstall

FIGURE 3-2: Submenus

NOTE *Not all plugins will need submenus. For example, a plugin with a single settings page has no need for additional submenus. When creating your new plugin, it's important to determine if submenus will be needed for a good user experience.*

Adding a Menu Item to an Existing Menu

If your plugin requires only a single options page, you may not need to create a custom top-level menu. Instead, you can simply add a submenu to an existing menu, such as the Settings menu.

WordPress features many different functions to add submenus to the existing default menus in WordPress. One of these functions is the `add_options_page()` function. Now explore how the `add_options_page()` function works to add a submenu item to the Settings menu.

```
<?php add_options_page( page_title, menu_title, capability, menu_slug, function); ?>
```

The `add_options_page()` function accepts the following parameters:

- `page_title`: Title of the page as shown in the `<title>` tags
- `menu_title`: Name of your submenu displayed on the Dashboard
- `capability`: Minimum capability required to view the submenu
- `menu_slug`: Slug name to refer to the submenu; should be a unique name
- `function`: Function to be called to display the page content for the item

Now add a submenu item to the Settings menu.

```
<?php
add_action( 'admin_menu', 'pdev_create_submenu' );

function pdev_create_submenu() {
```

```
    //create a submenu under Settings
    add_options_page( 'PDEV Plugin Settings', 'PDEV
Settings', 'manage_options',
    'pdev_plugin', 'pdev_plugin_option_page' );

?>
```

The preceding code adds a submenu labeled PDEV Settings under the Settings menu, as shown in [Figure 3-3](#). Set the page title to PDEV Plugin Settings, set the capability to manage_options so that only administrators can view it, and set the function pdev_plugin_option_page() to be called when the submenu is clicked.

 Dashboard

 Posts

 Media

 Pages

 Comments

 Appearance

 Plugins

 Users

 Tools

 Settings

General

Writing

Reading

Discussion

Media

Permalinks

Privacy

PDEV Settings

FIGURE 3-3: Submenu labeled PDEV Settings

The following is a list of all available submenu functions in WordPress:

- `add_dashboard_page`: Adds a submenu to the Dashboard menu
- `add_posts_page`: Adds a submenu to the Posts menu
- `add_media_page`: Adds a submenu to the Media menu
- `add_links_page`: Adds a submenu to the Links menu
- `add_pages_page`: Adds a submenu to the Pages menu
- `add_comments_page`: Adds a submenu to the Comments menu
- `add_theme_page`: Adds a submenu to the Appearance menu
- `add_plugins_page`: Adds a submenu to the Plugins menu
- `add_users_page`: Adds a submenu to the Users menu
- `add_management_page`: Adds a submenu to the Tools menu
- `add_options_page`: Adds a submenu to the Settings menu

To use any of these functions, simply swap out the function name in the code shown earlier.

NOTE *If your plugin requires only a single options page, it's best practice to add it as a submenu to an existing menu. If you require more than one, create a custom top-level menu.*

PLUGIN SETTINGS

Now that you've learned how to create menus and submenus in the WordPress Dashboard, it's time to create a settings page for your plugin. WordPress enables easy access to the database to store and retrieve data, such as options end users can modify and save in settings pages or internal information plugins you need to know. You'll learn how to save and fetch this data using the Options API and internal WordPress functions.

THE OPTIONS API

The Options API is a set of functions that enable easy access to the database where WordPress, plugins, and themes save and fetch needed information.

Options are stored in a database table named, by default, *wp_options* and can be text, integers, arrays, or objects. For example, WordPress keeps in this table the title of your blog, the list of active plugins, the news articles displayed on the Dashboard, or the time to check if a new version is available.

You'll now learn how to use the functions to access, update, and save options: `add_option()`, `update_option()`, `get_option()`, and `delete_option()`.

Saving Options

You start by saving your first plugin option, which will be named `pdev_plugin_color` and have a value of red. The function call to do so is the following:

```
<?php
add_option( 'pdev_plugin_color', 'red' );
?>
```

The `add_option()` function accepts the following parameters:

- `option`: Name of the option to add
- `value`: Value of the option you are adding
- `deprecated`: Description, which is no longer used
- `autoload`: Whether to load the option when WordPress starts

The first parameter is your option name. It is crucial that you make it unique and self-explanatory.

- **Unique:** It will never conflict with internal existing or future WordPress options or with settings that might be created by another plugin.

- **Self-explanatory:** Name it so that it's obvious it's a plugin setting and not something created by WordPress.

NOTE *Using the same prefix, for example, pdev_plugin, for function names, options, and variables is highly recommended for code consistency and for preventing conflict with other plugins. The golden rule of “Prefix everything,” first introduced in [Chapter 2](#), applies here.*

The second parameter is the option value that can be practically anything a variable can hold: string, integer, float number, Boolean, object, or array.

Saving an Array of Options

Every option saved adds a new record in WordPress' option table. You can simply store several options at once, in one array. This avoids cluttering the database and updates the values in a single MySQL query for greater efficiency and speed.

```
$options = array(  
    'color'    => 'red',  
    'fontsize' => '120%',  
    'border'   => '2px solid red'  
)  
  
update_option( 'pdev_plugin_options', $options );
```

Saving your plugin options in one array rather than individual records can have a huge impact on WordPress' loading time, especially if you save or update many options. Most of the time, PHP code executes fast, but SQL queries usually hinder performance, so save them whenever possible.

Updating Options

Now that you know how to create a new option for your plugin, let's look at updating that option value. To handle this, you'll use the `update_option()` function. As an example, let's update the color of your new plugin setting from red to blue.

```
<?php
update_option( 'pdev_plugin_color', 'blue' );
?>
```

The `update_option()` function accepts the following parameters:

- `option`: Name of the option to update
- `value`: Value of the option you are updating
- `autoload`: Whether to load the option when WordPress starts

The difference between `add_option()` and `update_option()` is that the first function does nothing if the option name already exists, whereas `update_option()` checks if the option already exists before updating its value and creates it if needed.

Retrieving Options

To fetch an option value from the database, use the function `get_option()`.

```
<?php
$pdev_plugin_color = get_option( 'pdev_plugin_color' );
?>
```

The first thing to know about `get_option()` is that if the option does not exist, it will return `false`. The second thing is that if you store Booleans, you might get integers in return.

The `get_option()` function accepts the following parameters:

- `option`: Name of the option to retrieve
- `default`: Value to return if the option does not exist. The default return value is `false`.

As an illustration of this behavior, consider the following code block that creates a couple of new options with various variable types:

```
<?php
update_option( 'pdev_bool_true', true );
update_option( 'pdev_bool_false', false );
?>
```

You can now retrieve these options, along with another one that does not exist, and see what variable types are returned, shown as an inline comment below each `get_option()` call:

```
<?php
var_dump( get_option( 'nonexistent_option' ) );
// bool(false)

var_dump( get_option( 'pdev_bool_true' ) );
// string(1) "1"

var_dump( get_option( 'pdev_bool_false' ) );
// bool(false)
?>
```

To avoid an error when checking option values, you should store `true` and `false` as `1` and `0`. This means also that you need to strictly compare the return of `get_option()` with Boolean `false` to check if the option exists.

```
<?php
//set the option as true
update_option( 'pdev_plugin_enabled', 1 );

if( get_option( 'pdev_plugin_enabled' ) == false ) {
    // option has not been defined yet
    // ...
    echo 'NOT DEFINED YET';
} else {
    // option exists
    // ...
    echo 'OPTION EXISTS!';
}
?>
```

You can also specify what value you want to be returned if the option is not found in the database, with a second option parameter to `get_option()`, like in the following example:

```
<?php
$option = get_option( 'pdev_plugin_option', 'Option not
found' );
?>
```

Loading an Array of Options

You have seen that saving multiple options in a single array is best practice. A complete example of saving and then getting values from one array would be as follows:

```
<?php
// To store all of them in a single function call:
$options = array(
    'color'      => 'red',
    'fontsize'   => '120%',
    'border'     => '2px solid red'
);

update_option( 'pdev_plugin_options', $options );

// Now to fetch individual values from one single call:
$options = get_option( 'pdev_plugin_options' );

// Store individual option values in variables
$color      = $options[ 'color' ];
$fontsize   = $options[ 'fontsize' ];
$border     = $options[ 'border' ];
?>
```

Saving and retrieving options enclosed in an array has another advantage: variable Boolean types within the array are preserved. Consider the following example:

```
<?php
add_option( 'pdev_test_bool', array(
    'booltrue'  => true,
    'boolfalse' => false
)
);
?>
```

Now get the option value from the database with `var_dump(get_option('test_bool'))`. See how Boolean types are retained, contrary to the previous example:

```
// output result
var_dump( get_option( 'pdev_test_bool' ) );

array(2) {
    ["booltrue"] => bool(true)
    ["boolfalse"]=> bool(false)
}
```

Deleting Options

Now that you understand how to create, update, and retrieve options, let's look at deleting an option. Deleting an option needs a self-explanatory function call.

```
<?php  
delete_option( 'pdev_plugin_options' );  
?>
```

This function call returns `false` if the option to delete cannot be found and returns `true` otherwise. You will mostly delete options when writing uninstall functions or files (see [Chapter 2](#)).

The Autoload Parameter

By default, all the options stored in the database are fetched by a single SQL query when WordPress initializes and then caches. This applies to internal WordPress core settings and options created and stored by plugins.

This is efficient behavior: no matter how many `get_option()` calls you issue in your plugins, they won't generate extra SQL queries and slow down the whole site. Still, the potential drawback of this autoload technique is that rarely used options are always loaded in memory, even when not needed. For instance, there is no point in fetching backend options when a reader accesses a blog post.

To address this issue when saving an option for the first time, you can specify its autoload behavior, as in the following example:

```
<?php  
add_option( 'pdev_plugin_option', $value, '', $autoload );  
?>
```

Note the empty third parameter: This is a parameter that was deprecated several WordPress versions ago and is not needed any more. Any value passed to it will do; just be sure not to omit it.

The fourth parameter is what matters here. If `$autoload` is anything but `'no'` (or simply not specified), option `pdev_plugin_option` will be read when WordPress starts, and subsequent `get_option()` function calls will not issue any supplemental SQL query. Setting `$autoload` to `'no'` can

invert this: this option will not be fetched during startup of WordPress, saving memory and execution time, but it will trigger an extra SQL query the first time your code fetches its value.

NOTE *If you want to specify the autoload parameter, you need to use add_option() instead of update_option() when creating an option the first time. If you don't need this parameter, always using update_option() to both create and update will make your code more simple and consistent.*

Of course, specifying the autoload parameter upon creation of an option does not change the way you fetch, update, or delete its value.

Segregating Plugin Options

A function to initiate your plugin options, run on plugin activation as covered in [Chapter 2](#), could then look like the following:

```
<?php
function pdev_plugin_create_options() {

    // front-end options: autoloaded
    add_option( 'pdev_plugin_options', array(
        'color'      => 'red',
        'fontsize'   => '120%',
        'border'     => '2px solid red'
    )
);

    // back-end options: loaded only if explicitly needed
    add_option( 'pdev_plugin_admin_options', array(
        'version'      => '1.0',
        'donate_url'   => 'https://example.com/',
        'advanced_options' => '1'
    ),
    '', 'no' );
}

?>
```

Again, don't forget the empty third parameter before the autoload value. This might seem a bit convoluted, and actually it is for so few options set.

This professional technique makes sense if your plugin features dozens of options, or options containing long text strings.

NOTE *As a rule of thumb, if your options are needed by the public part of the blog, save them with autoload. If they are needed only in the admin area, save them without autoload.*

Toggling the Autoload Parameter

The autoload parameter is set when an option is created with `add_option()` and is not supposed to change afterward. With this said, if you believe that it would improve your plugin's efficiency to modify the autoload behavior, it is possible and easy: simply delete and then re-create the option with an explicit autoload parameter.

```
<?php
function pdev_plugin_recreate_options() {

    // get old value
    $old = get_option( 'pdev_plugin_admin_options' );

    // delete then recreate without autoload
    delete_option( 'pdev_plugin_admin_options' );
    add_option( 'pdev_plugin_admin_options', $old, '', 'no'
);

}
?>
```

THE SETTINGS API

Options can be internally created and updated by your plugin (for instance, storing the time stamp of the next iteration of a procedure). But they are also frequently used to store settings the end user will modify through your plugin administration page.

When creating or updating user-defined options for a plugin, relying on the Settings API can make your code both simpler and more efficient.

Benefits of the Settings API

Dealing with user inputs introduces new constraints in the option process: you need to design a user interface, monitor form submissions, handle security checks, and validate user inputs. To easily manage these common tasks, WordPress wraps the option functions into a comprehensive Settings API.

The Settings API enables you to handle these simple tasks:

- Tell WordPress that you are going to use some new options and how you want them displayed.
- Specify a function that will sanitize user inputs.

while WordPress transparently manages for you these cumbersome and repetitive parts:

- Draw most of the option page itself.
- Monitor form submission and handle `$_POST` data.
- Create and update options if needed.
- Wield all the required security measures and hidden fields for nonces, covered in detail in [Chapter 4](#), “Security and Performance.”

Now it's time to dissect the Settings API; you'll learn to use it through a step-by-step example.

Settings API Functions

The Settings API functions consist of three steps:

1. Tell WordPress the new settings you want it to manage for you. Doing so adds your settings into a list of authorized options (also known as *whitelisting*).
2. Define the settings (text areas, input boxes, and any HTML form element) and how they will be visually grouped together in sections.
3. Tell WordPress to display your settings in an actual form.

But first, you create a setting management page for your plugin.

Creating the Plugin Administration Page

The plugin page will be located at Settings ⇔ PDev Settings:

```
<?php
// Add a menu for our option page
add_action( 'admin_menu', 'pdev_plugin_add_settings_menu' );

function pdev_plugin_add_settings_menu() {

    add_options_page( 'PDEV Plugin Settings', 'PDEV
Settings', 'manage_options',
    'pdev_plugin', 'pdev_plugin_option_page' );

}

// Create the option page
function pdev_plugin_option_page() {
    ?>
    <div class="wrap">
        <h2>My plugin</h2>
        <form action="options.php" method="post">
        </form>
    </div>
<?php
}
?>
```

This page is empty for now. You will add form inputs later. Creating pages for plugins is covered in detail earlier in the “Plugin Settings” section of this chapter, so refer to it for more explanation about this code.

Registering New Settings

The next step is to register your new settings. The function you need here is `register_setting()` and three parameters, used as follows:

```
<?php
register_setting( option group, option name, args );
?>
```

The `register_setting()` function accepts the following parameters:

- `option_group`: Group name for your settings.
- `option_name`: Name of an option to sanitize and save.

- args: Data used to describe the setting when registered.
 - type: Type of data associated with this setting. Valid values are string, boolean, integer, and number.
 - description: Description of the data for this setting.
 - sanitize_callback: Callback function to sanitize the option's value.
 - show_in_rest: Whether the data with this setting should be included in the REST API.
 - default: Default value when calling get_option().

Now let's register your plugin settings using the `register_setting()` function. The first parameter you'll set is the setting group name, and the second parameter is the option name as you would use it in a `get_option()` call. The group name can be anything actually, but it's just simpler to name it the same as the option that will get stored in the database.

The third parameter is an array of additional arguments defining the type of data. In this case, it's a string and references your callback function, here named `pdev_plugin_validate_options()`, that will be passed all the settings saved in your form. You'll define this function later.

```
<?php
$args = array(
    'type'          => 'string',
    'sanitize_callback' => 'pdev_plugin_validate_options',
    'default'        => NULL
);

register_setting( 'pdev_plugin_options',
    'pdev_plugin_options', $args );
?>
```

Defining Sections and Settings

Now define what the settings will be more precisely by using the function `add_settings_field()` and how they will be visually grouped with the function `add_settings_section()`.

```
<?php
add_settings_section( id, title, callback, page );
```

```
add_settings_field( id, title, callback, page, section, args
);
?>
```

The first function call, `add_settings_section()`, defines how the section on the page will show. The four required parameters it uses follow:

- `id`: HTML ID tag for the section
- `title`: Section title text that will show within an `<H2>` tag
- `callback`: Name of the callback function that will echo some explanations about that section
- `page`: Settings page on which to show the section (that is, the `?page=pdev_plugin` part of the page URL)

The second function call, `add_settings_field()`, describes how to add the form input. Its required parameters follow:

- `id`: HTML ID tag for the section
- `title`: Formatted title of the field, which is displayed as the label for the field on output
- `callback`: Name of the callback function that will echo the form field
- `page`: Settings page on which to show the section
- `section`: Section of the settings page in which to show the field, as defined previously by the `add_settings_section()` function call
- `args`: Additional arguments used when outputting the field

Now let's implement these new functions for your plugin settings as follows:

```
<?php
add_settings_section(
    'pdev_plugin_main',
    'PDEV Plugin Settings',
    'pdev_plugin_section_text',
    'pdev_plugin'
);
```

```
add_settings_field(
    'pdev_plugin_name',
    'Your Name',
    'pdev_plugin_setting_name',
    'pdev_plugin',
    'pdev_plugin_main'
);
```

You now need to define two simple callback functions: one to display a few explanations about the section and one to output and fill the form field.

```
<?php
// Draw the section header
function pdev_plugin_section_text() {

    echo '<p>Enter your settings here.</p>';

}

// Display and fill the Name form field
function pdev_plugin_setting_name() {

    // get option 'text_string' value from the database
    $options = get_option( 'pdev_plugin_options' );
    $name = $options['name'];

    // echo the field
    echo "<input id='name' name='pdev_plugin_options[name]' type='text' value='"
        . esc_attr( $name ) . "'>";

}

?>
```

This second function call fetches the option value name that is stored in an array.

When outputting the HTML input field, note its name. This is how you tell the browser to pass this value back into an array with the same name as the option you'll save, as defined earlier in the `register_setting()` function call. Any field that has not been previously registered and whitelisted will be ignored by WordPress.

You'll also notice you are using the `esc_attr()` function. This is used to escape the HTML attribute value for display. You'll learn more about this in

[Chapter 4.](#)

Validating User Input

There is still one callback function to define, `pdev_plugin_validate_options()`, as mentioned earlier when registering the settings.

In this example, users are asked to enter text, so your validation function simply makes sure that the input contains only letters.

```
<?php
// Validate user input (we want text and spaces only)
function pdev_plugin_validate_options( $input ) {

    $valid = array();
    $valid['name'] = preg_replace(
        '/[^a-zA-Z\s]/',
        '',
        $input['name'] );

    return $valid;
}

?>
```

To validate the user input as letters only, a simple pattern matching (also known as *regular expression*) that strips all other characters is used here. This regex pattern also supports spaces between names.

This function is passed the `$_POST` data as a parameter. For enhanced security, start creating a new empty array named `$valid` and collect in this array only the values you are expecting. This way, if for some reason an unanticipated field is submitted in the form, your function not only validates the information you want but also blocks everything else. Refer to [Chapter 4](#) for more tips and functions about data validation.

Rendering the Form

Now that you have defined these function calls, it's time to use them. At the beginning of this step-by-step example, you created an empty page. Go back to that and add the form fields and a Submit button.

```

// Create the option page
function pdev_plugin_option_page() {
?>
    <div class="wrap">
        <h2>My plugin</h2>
        <form action="options.php" method="post">
            <?php
            settings_fields( 'pdev_plugin_options' );
            do_settings_sections( 'pdev_plugin' );
            submit_button( 'Save Changes', 'primary' );
            ?>
        </form>
    </div>
<?php
}

```

The `settings_fields()` function call references the whitelisted option you have declared with `register_setting()`. The only required parameter is the settings group name. This should match the group name used in `register_setting()`. It takes care of the hidden fields, security checks, and form redirection after it has been submitted.

The second function call, `do_settings_sections()`, outputs all the sections and form fields you have previously defined. The only required parameter for this function is the slug name of the page whose settings sections you want to output.

The final function is `submit_button()`, which will display the form submission button. This function is extremely handy when building any form in WordPress and accepts the following parameters:

- `text`: Text displayed on the button.
- `type`: Type and CSS class or classes of the button.
- `name`: HTML name of the button.
- `wrap`: Boolean that defines whether the output button should be wrapped in a paragraph tag. Defaults to true.
- `other_attributes`: Additional attributes that should be output with the button, such as a `tabindex`.

An alternative method is using the `get_submit_button()` function. This function supports the same parameters, but instead of displaying the button, it will return the submit button. This is useful if you are compiling a form but not ready to fully display it yet.

All Done!

Notice how little HTML you have laid down, and yet the plugin page is now complete and functional. This is a major reason this Settings API is rock solid: you focus on features and let WordPress create all the HTML with relevant tags and classes, handle the data submission, and escape strings before inserting them to the database.

NOTE *Designing plugin pages using the Settings API is future-proof. Imagine that you are creating a plugin for a client on a particular version of WordPress. Later, when the administration interface of WordPress changes (different layout, colors, HTML classes), your plugin will still seamlessly integrate because you did not hard-code any HTML in it.*

Wrapping It Up: A Complete Plugin Management Page

Some of the function calls used here need to be hooked into WordPress actions such as `admin_init`. Let's recapitulate all the steps covered bit by bit into a full-fledged plugin.

```
<?php
/*
Plugin Name: Settings API example
Plugin URI: https://example.com/
Description: A complete and practical example of the
WordPress Settings API
Author: WROX
Author URI: http://wrox.com
*/
// Add a menu for our option page
add_action( 'admin_menu', 'pdev_plugin_add_settings_menu' );

function pdev_plugin_add_settings_menu() {
```

```
    add_options_page( 'PDEV Plugin Settings', 'PDEV
Settings', 'manage_options',
    'pdev_plugin', 'pdev_plugin_option_page' );

}

// Create the option page
function pdev_plugin_option_page() {
    ?>
    <div class="wrap">
        <h2>My plugin</h2>
        <form action="options.php" method="post">
            <?php
                settings_fields( 'pdev_plugin_options' );
                do_settings_sections( 'pdev_plugin' );
                submit_button( 'Save Changes', 'primary' );
            ?>
        </form>
    </div>
    <?php
}

// Register and define the settings
add_action( 'admin_init', 'pdev_plugin_admin_init' );

function pdev_plugin_admin_init(){
    $args = array(
        'type' => 'string',
        'sanitize_callback' =>
'pdev_plugin_validate_options',
        'default' => NULL
    );
}

// Register our settings
register_setting( 'pdev_plugin_options',
'pdev_plugin_options', $args );

// Add a settings section
add_settings_section(
    'pdev_plugin_main',
    'PDEV Plugin Settings',
    'pdev_plugin_section_text',
    'pdev_plugin'
);
```

```

    // Create our settings field for name
    add_settings_field(
        'pdev_plugin_name',
        'Your Name',
        'pdev_plugin_setting_name',
        'pdev_plugin',
        'pdev_plugin_main'
    );
}

// Draw the section header
function pdev_plugin_section_text() {

    echo '<p>Enter your settings here.</p>';
}

// Display and fill the Name form field
function pdev_plugin_setting_name() {

    // get option 'text_string' value from the database
    $options = get_option( 'pdev_plugin_options' );
    $name = $options['name'];

    // echo the field
    echo "<input id='name' name='pdev_plugin_options[name]' type='text' value='"
        . esc_attr( $name ) . "'/>";
}

// Validate user input (we want text and spaces only)
function pdev_plugin_validate_options( $input ) {

    $valid = array();
    $valid['name'] = preg_replace(
        '/[^a-zA-Z\s]/',
        '',
        $input['name'] );
}

return $valid;
}

?>

```

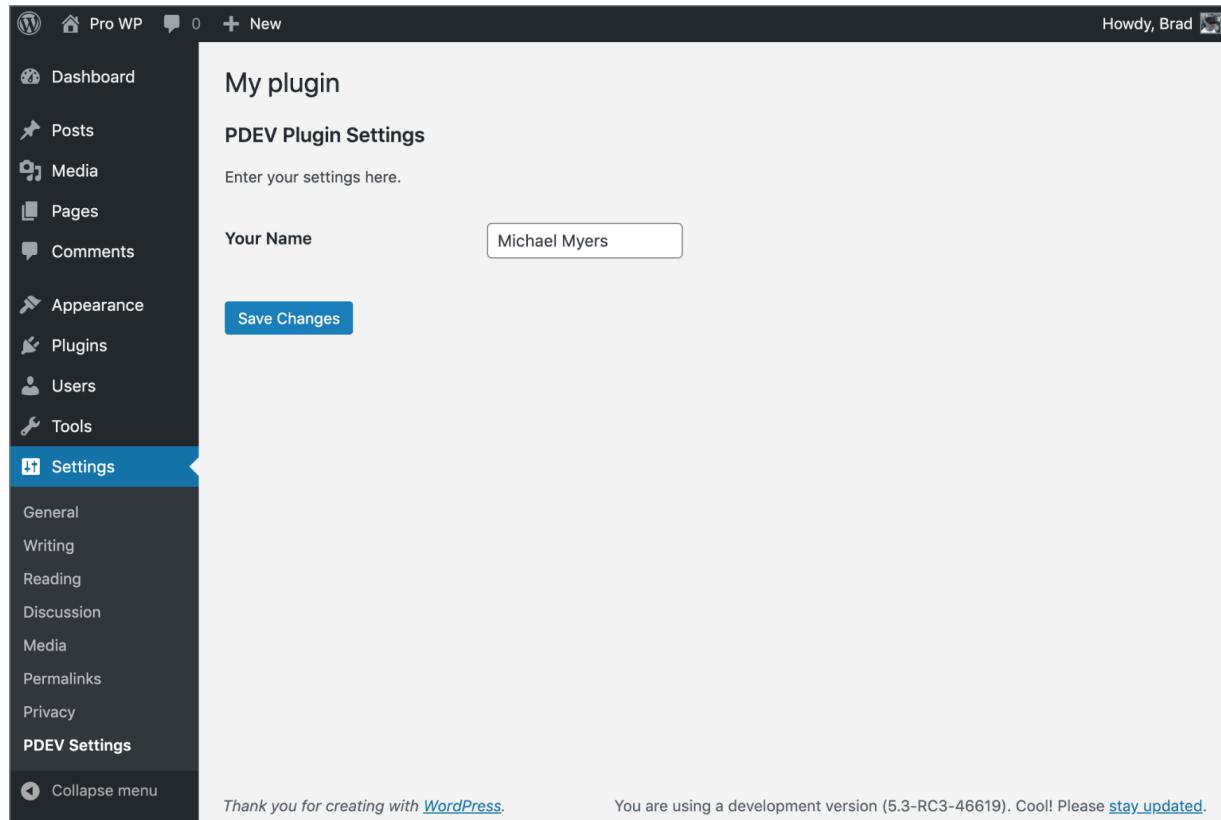


FIGURE 3-4: Plugin management page

Activate this plugin and head to Settings \Rightarrow PDEV Settings. You'll see a similar interface to the one shown in [Figure 3-4](#).

Improving Feedback on Validation Errors

The validation function you've previously defined could be slightly improved by letting the users know they have entered an unexpected value and that it has been modified so that they can pay attention to it and maybe amend their input.

The relatively unknown function `add_settings_error()` of the Settings API can handle this case. Here's how it is used:

```
<?php
add_settings_error( $setting, $code, $message, $type );
?>
```

This function call registers an error message that displays to the user. `add_settings_error()` accepts the following parameters:

- setting: Title of the setting to which this error applies
- code: Slug name to identify the error, which is used as part of the HTML ID tag
- message: Formatted message text to display, which is displayed inside styled <div> and <p> tags
- type: Type of message, error or update

You can improve the validating function with a user notice if applicable, as shown here:

```

<?php
function pdev_plugin_validate_options( $input ) {

    $valid['name'] = preg_replace(
        '/[^a-zA-Z\s]/',
        '',
        $input['name'] );

    if( $valid['name'] !== $input['name'] ) {

        add_settings_error(
            'pdev_plugin_text_string',
            'pdev_plugin_texterror',
            'Incorrect value entered! Please only input
letters and spaces.',
            'error'
        );
    }

    return $valid;
}
?>

```

The function now compares the validated data with the original input and displays an error message if they differ (see [Figure 3-5](#)).

My plugin

Incorrect value entered! Please only input letters and spaces.



PDEV Plugin Settings

Enter your settings here.

Your Name

Michael Myers

Save Changes

FIGURE 3-5: Error message

Expanding with Additional Field Types

You've now covered the different ways to create a settings page using the WordPress Settings API. Let's expand on that by adding additional form field types to your settings page. This is a more real-world example showing how elaborate your settings page could be. At the end of this section, you'll review the entire plugin code as a whole.

First, you need to define additional settings fields using the `add_settings_field()` function. The following example updates the registration of those fields in your custom `pdev_plugin_admin_init()` function:

```
// Register and define the settings
add_action('admin_init', 'pdev_plugin_admin_init');

function pdev_plugin_admin_init(){

    //define the setting args
    $args = array(
```

```
        'type' => 'string',
        'sanitize_callback' =>
'pdev_plugin_validate_options',
        'default' => NULL
    );

    //register our settings
    register_setting( 'pdev_plugin_options',
'pdev_plugin_options', $args );

    //add a settings section
    add_settings_section(
        'pdev_plugin_main',
        'PDEV Plugin Settings',
        'pdev_plugin_section_text',
        'pdev_plugin'
    );

    //create our settings field for name
    add_settings_field(
        'pdev_plugin_name',
        'Your Name',
        'pdev_plugin_setting_name',
        'pdev_plugin',
        'pdev_plugin_main'
    );

    //create our settings field for favorite holiday
    add_settings_field(
        'pdev_plugin_fav_holiday',
        'Favorite Holiday',
        'pdev_plugin_setting_fav_holiday',
        'pdev_plugin',
        'pdev_plugin_main'
    );

    //create our settings field for beast mode
    add_settings_field(
        'pdev_plugin_beast_mode',
        'Enable Beast Mode?',
        'pdev_plugin_setting_beast_mode',
        'pdev_plugin',
        'pdev_plugin_main'
    );

}
```

As you can see in the preceding code sample, you register two new form fields for Favorite Holiday and Beast Mode. Both call custom functions for displaying each field on your settings page. Let's look at the `pdev_plugin_setting_fav_holiday()` function first.

```
// Display and select the favorite holiday select field
function(pdev_plugin_setting_fav_holiday) {

    // Get option 'fav_holiday' value from the database
    // Set to 'Halloween' as a default if the option does
    not exist
    $options = get_option( 'pdev_plugin_options', [
    'fav_holiday' => 'Halloween' ] );
    $fav_holiday = $options['fav_holiday'];

    // Define the select option values for favorite holiday
    $items = array( 'Halloween', 'Christmas', 'New Years' );

    echo "<select id='fav_holiday'
name='pdev_plugin_options[fav_holiday]'>";

    foreach( $items as $item ) {

        // Loop through the option values
        // If saved option matches the option value, select
        it
        echo "<option value=\"" . $item. "\" "
            . selected( $fav_holiday, $item, false ).">" .
        esc_html( $item ) .
            "</option>";

    }

    echo "</select>";

}
```

This example registers a new select drop-down field allowing the user to select their favorite holiday. First, you call `get_option()` to load the current setting value, if it exists. If the value does not exist, meaning the user has not selected and saved the selection yet, the default is set to Halloween. Next you set the `$items` variable to an array of available holiday options to choose from.

Now it's time to loop through the available holiday options and display each one in the drop-down field. We're using the WordPress core function `selected()`, which compares two values and, if they match, returns `selected="selected"`, which will select that option value in the drop-down.

Now let's look at the custom function for displaying the radio button field, `pdev_plugin_setting_beast_mode()`.

```
// Display and set the Beast Mode radio button field
function pdev_plugin_setting_beast_mode() {

    // Get option 'beast_mode' value from the database
    // Set to 'disabled' as a default if the option does not
    // exist
    $options = get_option( 'pdev_plugin_options', [
        'beast_mode' => 'disabled' ] );
    $beast_mode = $options['beast_mode'];

    // Define the radio button options
    $items = array( 'enabled', 'disabled' );

    foreach( $items as $item ) {

        // Loop the two radio button options and select if
        // set in the option value
        echo "<label><input ".checked( $beast_mode, $item,
        false )."
            value= '" . esc_attr( $item ) . "'"
            name='pdev_plugin_options[beast_mode]'"
            type='radio' />" . esc_html( $item ) . "</label>
<br/>";

    }
}
```

This example registers a new radio button field allowing the user to select if they want to enable beast mode or not. Again, you'll use `get_option()` to load the current setting value, and if it doesn't exist, the default value `disabled` is set. Next, you'll define the two values in the `$items` array. Now you need to loop through the two options displaying the radio button. The `checked()` function is used to compare the saved value against the display

value and, if they are the same, to input checked="checked" on the radio button form field.

Now that you have registered two new settings fields and created the custom functions to display both fields, you need to update your validate function to accept those values and sanitize them as needed.

```
// Validate user input for all three options
function pdev_plugin_validate_options( $input ) {

    // Only allow letters and spaces for the name
    $valid['name'] = preg_replace(
        '/[^a-zA-Z\s]/',
        '',
        $input['name'] );

    if( $valid['name'] !== $input['name'] ) {

        add_settings_error(
            'pdev_plugin_text_string',
            'pdev_plugin_texterror',
            'Incorrect value entered! Please only input
letters and spaces.',
            'error'
        );
    }

    // Sanitize the data we are receiving
    $valid['fav_holiday'] = sanitize_text_field(
    $input['fav_holiday'] );
    $valid['beast_mode'] = sanitize_text_field(
    $input['beast_mode'] );
    return $valid;
}
```

The only update here is to run the two new option values through the `sanitize_text_field()` WordPress function to sanitize the user-inputted data. Even though the values are hard-coded in the form, that doesn't stop a user from modifying the frontend form code and posting unknown data to these setting fields. You'll cover this more in [Chapter 4](#).

That's it! You now have a more robust settings page for your plugin. Let's review the entire plugin source code with your new settings fields.

```
<?php
/*
Plugin Name: Settings API example - Multiple Fields
Plugin URI: https://example.com/
Description: A complete and practical example of the
WordPress Settings API
Author: WROX
Author URI: http://wrox.com
*/
// Add a menu for our option page
add_action( 'admin_menu', 'pdev_plugin_add_settings_menu' );

function pdev_plugin_add_settings_menu() {

    add_options_page( 'PDEV Plugin Settings', 'PDEV
Settings', 'manage_options',
    'pdev_plugin', 'pdev_plugin_option_page' );

}

// Create the option page
function pdev_plugin_option_page() {
    ?>
    <div class="wrap">
        <h2>My plugin</h2>
        <form action="options.php" method="post">
            <?php
            settings_fields( 'pdev_plugin_options' );
            do_settings_sections( 'pdev_plugin' );
            submit_button( 'Save Changes', 'primary' );
            ?>
        </form>
    </div>
    <?php
}

// Register and define the settings
add_action( 'admin_init', 'pdev_plugin_admin_init' );

function pdev_plugin_admin_init(){

    // Define the setting args
    $args = array(
        'type'              => 'string',
        'sanitize_callback' =>
    'pdev_plugin_validate_options',
        'default'           => NULL
}
```

```
);

// Register our settings
register_setting( 'pdev_plugin_options',
'pdev_plugin_options', $args );

// Add a settings section
add_settings_section(
    'pdev_plugin_main',
    'PDEV Plugin Settings',
    'pdev_plugin_section_text',
    'pdev_plugin'
);

// Create our settings field for name

add_settings_field(
    'pdev_plugin_name',
    'Your Name',
    'pdev_plugin_setting_name',
    'pdev_plugin',
    'pdev_plugin_main'
);

// Create our settings field for favorite holiday

add_settings_field(
    'pdev_plugin_fav_holiday',
    'Favorite Holiday',
    'pdev_plugin_setting_fav_holiday',
    'pdev_plugin',
    'pdev_plugin_main'
);

// Create our settings field for beast mode

add_settings_field(
    'pdev_plugin_beast_mode',
    'Enable Beast Mode?',
    'pdev_plugin_setting_beast_mode',
    'pdev_plugin',
    'pdev_plugin_main'
);

}

// Draw the section header
```

```

function pdev_plugin_section_text() {
    echo '<p>Enter your settings here.</p>';
}

// Display and fill the Name text form field
function pdev_plugin_setting_name() {
    // Get option 'text_string' value from the database
    $options = get_option( 'pdev_plugin_options' );
    $name = $options['name'];

    // Echo the field
    echo "<input id='name' name='pdev_plugin_options[name]' type='text' value='" . esc_attr( $name ) . "'/>";
}

// Display and select the favorite holiday select field
function pdev_plugin_setting_fav_holiday() {
    // Get option 'fav_holiday' value from the database
    // Set to 'Halloween' as a default if the option does
    not exist
    $options = get_option( 'pdev_plugin_options', [
        'fav_holiday' => 'Halloween' ] );
    $fav_holiday = $options['fav_holiday'];

    // Define the select option values for favorite holiday
    $items = array( 'Halloween', 'Christmas', 'New Years' );

    echo "<select id='fav_holiday' name='pdev_plugin_options[fav_holiday]'>";
}

foreach( $items as $item ) {
    // Loop through the option values
    // If saved option matches the option value, select
    it
}

```

```

        echo "<option value=\"" . esc_attr( $item ) . "'"
            ".selected( $fav_holiday, $item, false ).">" .
esc_html( $item ) .
        "</option>";
    }

    echo "</select>";

}

// Display and set the Beast Mode radio button field

function pdev_plugin_setting_beast_mode() {

    // Get option 'beast_mode' value from the database
    // Set to 'disabled' as a default if the option does not
exist
    $options = get_option( 'pdev_plugin_options', [
'beast_mode' => 'disabled' ] );
    $beast_mode = $options['beast_mode'];

    // Define the radio button options
    $items = array( 'enabled', 'disabled' );

    foreach( $items as $item ) {

        // Loop the two radio button options and select if
set in the option value
        echo "<label><input " . checked( $beast_mode, $item,
false ) . "
            value='" . esc_attr( $item ) . "'"
name='pdev_plugin_options[beast_mode]'"
            type='radio' /> " . esc_html( $item ) . "</label>
<br/>";

    }

}

// Validate user input for all three options
function pdev_plugin_validate_options( $input ) {

    // Only allow letters and spaces for the name
    $valid['name'] = preg_replace(
        '/[^a-zA-Z\s]/',
        '',

```

```

    $input['name'] );

    if( $valid['name'] !== $input['name'] ) {

        add_settings_error(
            'pdev_plugin_text_string',
            'pdev_plugin_texterror',
            'Incorrect value entered! Please only input
letters and spaces.',
            'error'
        );

    }

    // Sanitize the data we are receiving
    $valid['fav_holiday'] = sanitize_text_field(
$input['fav_holiday'] );
    $valid['beast_mode'] = sanitize_text_field(
$input['beast_mode'] );

    return $valid;
}
?>

```

Adding Fields to an Existing Page

You have seen how to create a new custom settings page for a plugin and its associated entry in the administration menus. Doing so makes sense if your plugin features a lot of settings and its administration page shows a lot of content.

Sometimes, though, it is not worth adding a new menu entry for just one or a few plugin options. Here again the Settings API will prove to be useful, allowing plugin setting fields to easily be added to the existing WordPress setting pages.

How It Works

Two internal functions, `do_settings_sections()` and `do_settings_fields()`, are triggered to draw sections and fields that have been previously registered, like you did in the example plugin.

Each core setting page calls these two functions, so you can hook into them if you know their slug name.

Adding a Section to an Existing Page

Your previous plugin was adding a whole new section and its input field on a stand-alone page. You now modify it to insert this content into WordPress' Privacy Settings page.

```
<?php
$args = array(
    'type'          => 'string',
    'sanitize_callback' => 'pdev_plugin_validate_options',
    'default'        => NULL
);

register_setting( 'reading', 'pdev_plugin_options', $args );

add_settings_section(
    'pdev_plugin_options',
    'PDEV Plugin Settings',
    'pdev_plugin_section_text',
    'reading'
);

add_settings_field(
    'pdev_plugin_text_string',
    'Your Name',
    'pdev_plugin_setting_name',
    'reading',
    'pdev_plugin_options'
);
?>
```

Notice that the first parameter passed in the `register_setting()` function call is set to `reading`. This function now adds your custom section into the `reading` section, which is located within the Reading Settings page, as shown in [Figure 3-6](#). Replace all `reading` instances with `media`, and your section will be appended at the end of the Media Settings page.

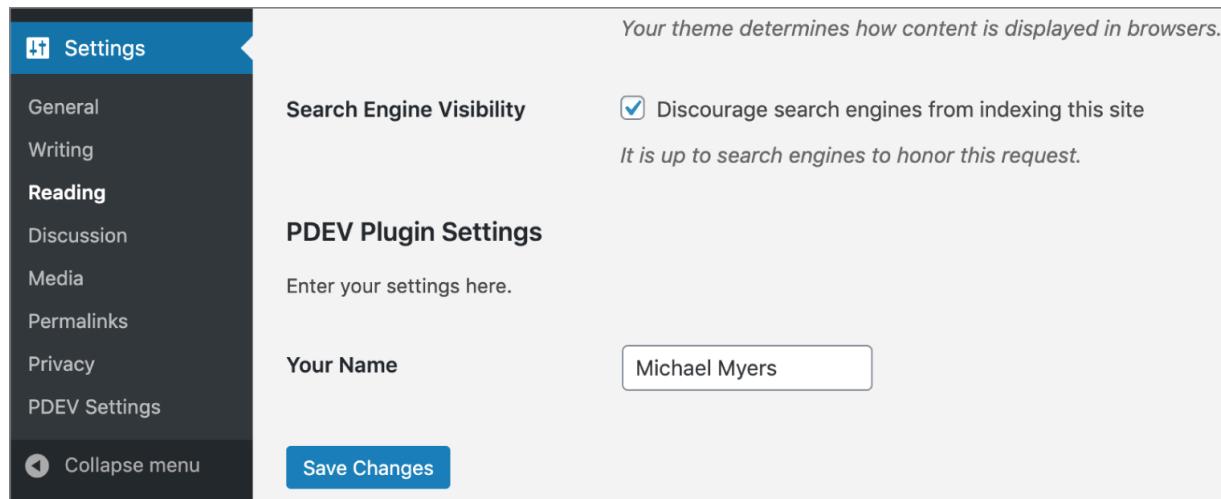


FIGURE 3-6: Section appended

You still need to whitelist this setting, with `register_setting()`. Omitting this step would make WordPress ignore the setting when submitting the form.

Adding Only Fields

Of course, it can even make sense to add just one field and no section header to an existing page. Now modify the function in the previous example as shown here:

```
<?php
function pdev_plugin_admin_init(){

    $args = array(
        'type'          => 'string',
        'sanitize_callback' =>
    'pdev_plugin_validate_options',
        'default'        => NULL
    );

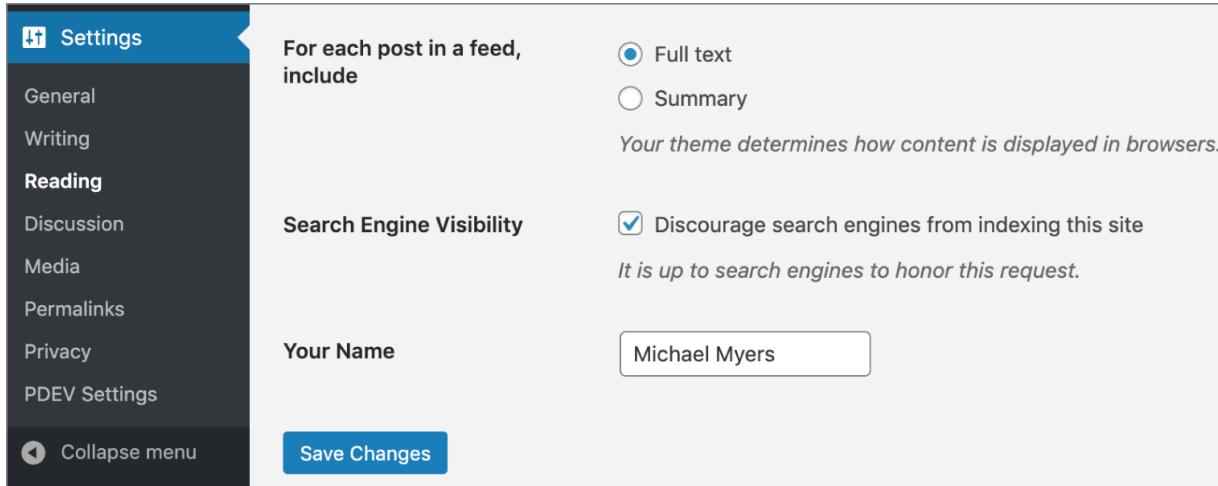
    register_setting( 'reading', 'pdev_plugin_options',
    $args );

    add_settings_field(
        'pdev_plugin_text_string',
        'Your Name',
        'pdev_plugin_setting_name',
        'reading',
        'default'
    );
}
```

```
}
```

```
?>
```

Your singular field will be added to the default field set of the reading section, as shown in [Figure 3-7](#).



[FIGURE 3-7](#): Singular field

WordPress' Sections and Setting Fields

To add a section to an existing page or a field to an existing section, all you need to know is the slug name of the page. [Table 3-1](#) includes every section and field set names found in the WordPress Settings pages.

TABLE 3-1: List of Core Sections and Fields

WORDPRESS' SETTINGS PAGES	SECTION NAMES	FIELD SET NAMES
General Settings (options-general.php)	general	default
Writing Settings (options-writing.php)	writing	default remote_publishing post_via_email
Reading Settings (options-reading.php)	reading	default
Discussion Settings (options-discussion.php)	discussion	default avatars
Media Settings (options-media.php)	media	default embeds uploads
Permalink Settings (options-permalink.php)	permalink	optional

User Interface Concerns

Electing to add your plugin settings to a separate page or to a core WordPress page is often a matter of choosing the right user interface for the right end user.

When working on a site for a client, you may focus on delivering a key-in-hand CMS solution and not on explaining what is WordPress and what is a plugin extending its features. Adding your plugin settings to a core Settings page can enhance its integration into WordPress' backend because it won't appear different from other core settings. From the client's point of view, your plugin is a core element just as any other built-in feature.

On the contrary, if you intend to make your plugin available for download, you can target people who probably understand the concept of adding new features to a core set. These people will naturally search for a custom menu where they can manage your plugin. If you opt for adding fields to an

existing page, be sure to explicitly tell users about it, for instance in the plugin documentation.

Removing Settings

As a professional plugin developer, it's important to create a high-quality experience for your users, which includes tasks that are rarely noticed. In this case you're going to learn how to remove settings that your plugin has created when it is uninstalled in WordPress.

You're going to use `register_uninstall_hook()`, which was covered in [Chapter 2](#), “Plugin Framework.” First you need to register your uninstall function as shown here:

```
register_uninstall_hook( __FILE__, 'pdev_plugin_uninstall'  
);
```

Next you'll use the `unregister_setting()` function of the Settings API. The `unregister_setting()` function accepts the following parameters:

- `option_group`: Settings group name used during registration
- `option_name`: Name of the option to unregister

You can simply grab these two values from when you registered your settings group. Let's look at the full function in action.

```
// Deregister our settings group and delete all options  
function pdev_plugin_uninstall() {  
  
    // Clean de-registration of registered setting  
    unregister_setting( 'pdev_plugin_options',  
    'pdev_plugin_options' );  
  
    // Remove saved options from the database  
    delete_option( 'pdev_plugin_options' );  
  
}
```

The `unregister_setting()` function will unregister your settings group from WordPress. You also need to remove the value from the database using `delete_option()`. Remember you are using the uninstall hook, which means this code will be executed only if the plugin is uninstalled. If the

plugin is deactivated, the options will remain and be available should the user decide to reactivate the plugin.

KEEPING IT CONSISTENT

They say consistency is one of the principles of good UI design. Creating a plugin for WordPress is no different, and it's a best practice to make your plugin match the WordPress user interface as closely as possible. This helps keep the interface consistent for end users and can make your plugin more professional by providing a solid user experience from the start.

One of the primary advantages to using the WordPress Settings API is that the UI design elements are handled for you. The headers, description text, form fields, buttons, and notices are all styled exactly as the rest of the WordPress Dashboard. It's also future-proof, meaning if the WordPress admin design and styles are updated in a future version, your plugin will automatically use the updated styling.

WordPress features many different styles that you can easily use in your plugin. In this section, you'll learn how to use the styling available in WordPress for your plugins. To demonstrate, create a simple plugin with a settings page.

```
<?php
add_action( 'admin_menu', 'pdev_styling_create_menu' );

function pdev_styling_create_menu() {

    //create custom top-level menu
    add_menu_page( 'Testing Plugin Styles', 'Plugin
Styling',
        'manage_options', __FILE__, 'pdev_styling_settings'
);
}

?>
```

Throughout this section, you'll modify the `pdev_styling_settings()` function.

Using the WordPress UI

The most important part of using the WordPress styles is to wrap your plugin in the class `wrap` `div`.

```
<div class="wrap">
    Plugin Page
</div>
```

This class sets the stage for all admin styling.

Headings

WordPress has custom styles available for all heading tags. Now look at how those heading tags display:

```
<?php
function pdev_styling_settings() {
    ?
    <div class="wrap">
        <h1>My Plugin</h1>
        <h2>My Plugin</h2>
        <h3>My Plugin</h3>
        <h4>My Plugin</h4>
        <h5>My Plugin</h5>
        <h6>My Plugin</h6>
    </div>
    <?php
}
?>
```

Each heading is slightly smaller than the previous, as shown in [Figure 3-8](#). The main heading should always be an `<h1>`, and any subsequent headings should (likely) be bumped one level up. There should be no skipped levels. This is important for accessibility support as most screen readers will start with the `<h1>` tag and progress down from there.

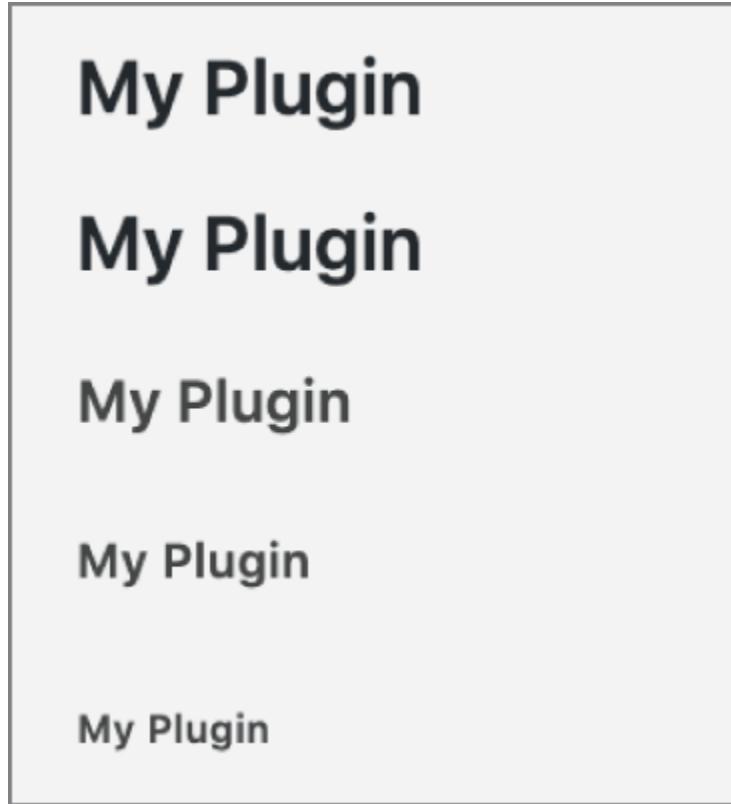


FIGURE 3-8: Heading levels

Dashicons

WordPress features its own open source font icon library called Dashicons. These icons (or *dashicons*) are used throughout the WordPress Dashboard and are available to use easily in your plugins.

As an example, let's add some fun dashicons to your various plugin headers.

```
<h1><span class="dashicons dashicons-smiley"></span> My  
Plugin</h1>  
<h2><span class="dashicons dashicons-visibility"></span> My  
Plugin</h2>  
<h3><span class="dashicons dashicons-universal-access">  
</span> My Plugin</h3>  
<h4><span class="dashicons dashicons-buddicons-replies">  
</span> My Plugin</h4>  
<h5><span class="dashicons dashicons-businesswoman"></span>  
My Plugins</h5>  
<h6><span class="dashicons dashicons-thumbs-up"></span> My  
Plugin</h6>
```

You simply add a custom dashicons class to your `` tag to identify the dashicon you'd like to display. In this example you're using a `span` tag, but they can be added to additional HTML tags as well, such as a `paragraph` tag. The preceding code generates the icons shown in [Figure 3-9](#).

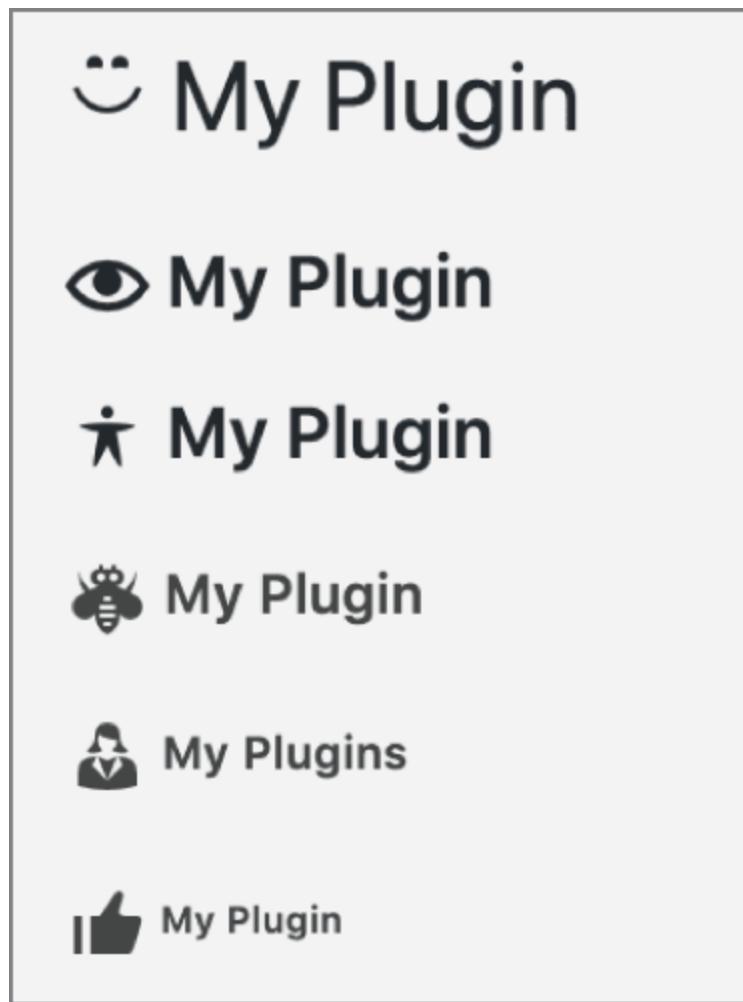


FIGURE 3-9: Dashicons

It's important to note that dashicons are automatically loaded within the WordPress Dashboard, but not on the frontend of the website. If you'd like to use dashicons on the public side, you'll need to enqueue the dashicon script, as shown here:

```
<?php
add_action( 'wp_enqueue_scripts',
'pdev_load_dashicons_front_end' );

function pdev_load_dashicons_front_end() {
```

```
        wp_enqueue_style( 'dashicons' );
    }
?>
```

Now your plugin has a clean header and uses the Plug icon.

For more information and a complete list of all Dashicons available, see the official resource at

<https://developer.wordpress.org/resource/dashicons>.

Messages

When an action occurs in your plugin, such as saving settings, it's important to display a message to the user verifying whether the update was successful. WordPress features some different styles for displaying these messages.

```
<?php
function pdev_styling_settings() {
    ?>
    <div class="notice notice-error is-dismissible">
        <p>There has been an error.</p>
    </div>

    <div class="notice notice-warning is-dismissible">
        <p>This is a warning message.</p>
    </div>

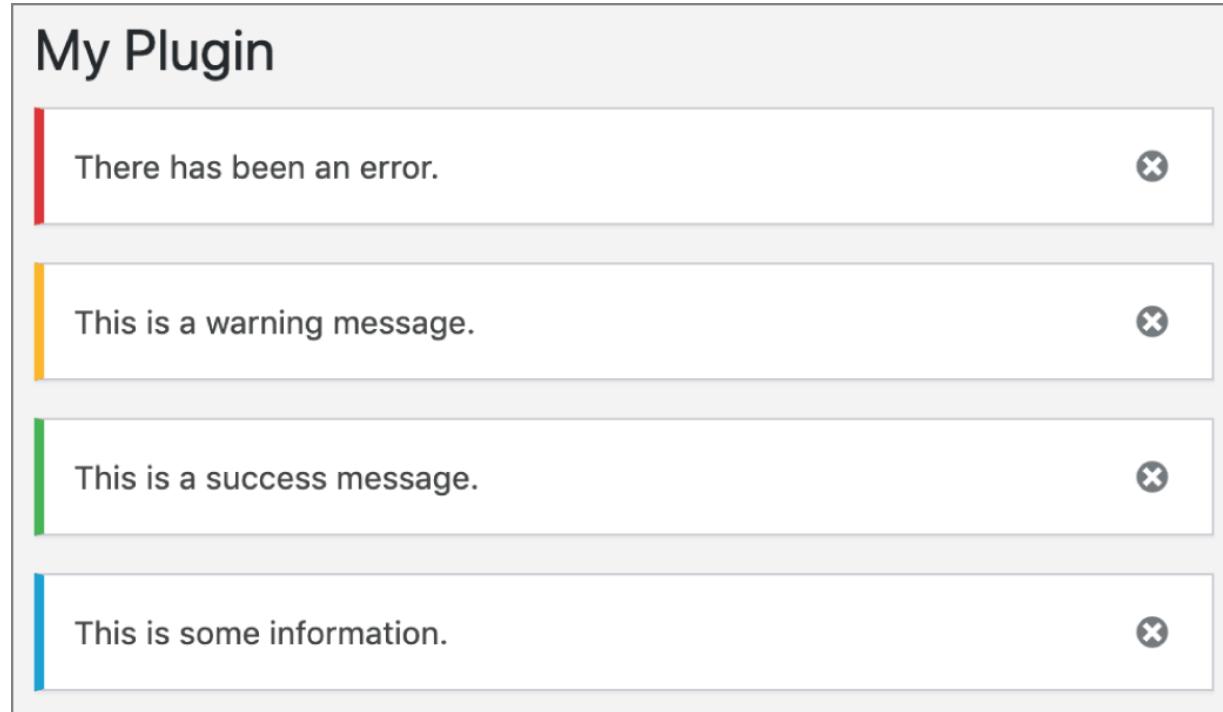
    <div class="notice notice-success is-dismissible">
        <p>This is a success message.</p>
    </div>

    <div class="notice notice-info is-dismissible">
        <p>This is some information.</p>
    </div>
    <?php
}
?>
```

As you can see, there are four different types of notices supported: error, warning, success, and info. Notice the class, `is-dismissible`, included in the outer `<div>` element. This class makes the admin notice “dismissible,” meaning a small `x` is displayed, and when it's clicked, the notice will

disappear. When an admin notice is not dismissible, the only way to remove it is to reload the page.

These styles will generate the messages shown in [Figure 3-10](#).



[FIGURE 3-10](#): Dismissible notices

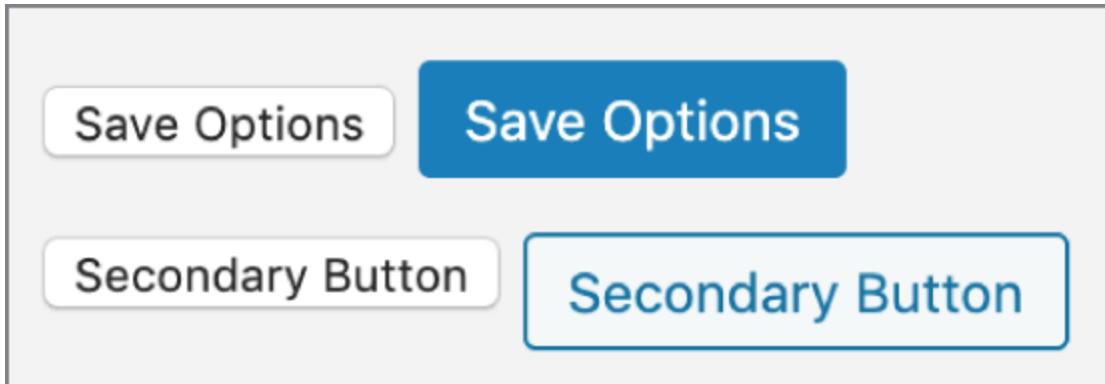
Buttons

As discussed earlier, the easiest method for adding a form submission button is using the `submit_button()` function. However, there's no reason you can't manually create form buttons using the preset WordPress admin stylings. When manually adding buttons to your form, you can take advantage of multiple classes. The first two you use are the `button-primary` and `button-secondary` classes. These classes style your buttons to match the WordPress UI.

```
<p>
<input type="submit" name="Save" value="Save Options"/>
<input type="submit" name="Save" value="Save Options"
       class="button-primary"/>
</p><p>
<input type="submit" name="Secondary" value="Secondary
Button"/>
```

```
<input type="submit" name="Secondary" value="Secondary  
Button"  
    class="button-secondary"/>  
</p>
```

This example demonstrates a standard unstyled button as compared to the WordPress styled button. As you can tell, the WordPress-styled button looks familiar and uses the proper styling, as shown in [Figure 3-11](#).

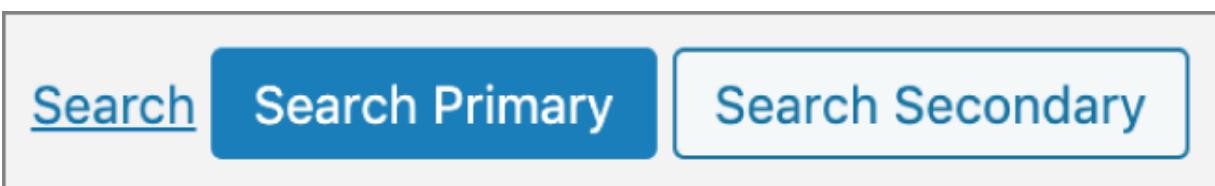


[FIGURE 3-11](#): WordPress-styled button

Links can also take the form of a button by using the appropriate class.

```
<a href="#">Search</a>  
<a href="#" class="button-primary">Search Primary</a>  
<a href="#" class="button-secondary">Search Secondary</a>
```

This example shows how a standard `<a href>` link can be styled to look like a button, as shown in [Figure 3-12](#). To normal users, they would never know these are regular text links because they look just like a button.



[FIGURE 3-12](#): Link styled to look like a button

Form Fields

WordPress has a special table class just for forms called `form-table`. This class is used on all WordPress Dashboard forms, including every Settings

page. This is a useful class when creating any type of options in your plugin.

```
<div class="wrap">
    <?php screen_icon( 'plugins' ); ?>
    <h2>My Plugin</h2>
    <form method="POST" action="">
        <table class="form-table">
            <tr valign="top">
                <th scope="row"><label for="fname">First Name</label></th>
                <td><input maxlength="45" size="25" name="fname"/></td>
            </tr>
            <tr valign="top">
                <th scope="row"><label for="lname">Last Name</label></th>
                <td><input id="lname" maxlength="45" size="25" name="lname"/></td>
            </tr>
            <tr valign="top">
                <th scope="row"><label for="color">Favorite Color</label></th>
                <td>
                    <select name="color">
                        <option value="orange">Orange</option>
                        <option value="black">Black</option>
                    </select>
                </td>
            </tr>
            <tr valign="top">
                <th scope="row"><label for="featured">Featured?</label></th>
                <td><input type="checkbox" name="favorite"/>
            </td>
            </tr>
            <tr valign="top">
                <th scope="row"><label for="gender">Gender</label></th>
                <td>
                    <input type="radio" name="gender" value="male"/> Male
                    <input type="radio" name="gender" value="female"/> Female
                </td>
            </tr>
            <tr valign="top">
```

```
        <th scope="row"><label for="bio">Bio</label>
    </th>
        <td><textarea name="bio"></textarea></td>
    </tr>
    <tr valign="top">
        <td>
            <input type="submit" name="save" value="Save
Options"
                class="button-primary"/>
            <input type="submit" name="reset" value="Reset"
                class="button-secondary"/>
        </td>
    </tr>
    </table>
    </form>
</div>
```

Using the `form-table` can give your options a familiar look to your plugin users. This makes for a better user experience, as shown in [Figure 3-13](#).

First Name	Brad
Last Name	Williams
Favorite Color	Orange ▾
Featured?	<input checked="" type="checkbox"/>
Gender	<input checked="" type="radio"/> Male <input type="radio"/> Female
Bio	<div><p>This is my bio. There are many like it but this one is mine.</p></div>
Save Options Reset	

FIGURE 3-13: WordPress-like options

Tables

HTML tables are a great way to display rows and columns of data in an easy-to-read layout. Tables can easily be styled in WordPress using the `widefat` class.

```

<table class="widefat">
<thead>
  <tr>
    <th>Name</th>
    <th>Favorite Holiday</th>
  </tr>
</thead>
<tfoot>
  <tr>
    <th>Name</th>
    <th>Favorite Holiday</th>
  </tr>
</tfoot>
<tbody>
  <tr>
    <td>Brad Williams</td>
    <td>Halloween</td>
  </tr>
  <tr>
    <td>Ozh Richard</td>
    <td>Talk Like a Pirate</td>
  </tr>
  <tr>
    <td>Justin Tadlock</td>
    <td>Christmas</td>
  </tr>
</tbody>
</table>

```

The `widefat` class has specific styles set for the `thead` and `tfoot` HTML tags. This styles the header and footer of your table to match all other tables on the Dashboard. The class can also style all table data, as shown in [Figure 3-14](#).

Name	Favorite Holiday
Brad Williams	Halloween
Justin Tadlock	Christmas
John James Jacoby	Talk Like a Pirate
Name	Favorite Holiday

FIGURE 3-14: Table style

Pagination

If your plugin contains a list of records, you may have a need for pagination, which is the method to break lists of data into multiple pages and have links to load each individual page. This helps reduce the load times and makes it a much cleaner user experience to navigate through the data. Would you rather view 500 records on a page or 10 pages with 50 records on each page?

WordPress has a few different classes to style your pagination. The following is an example:

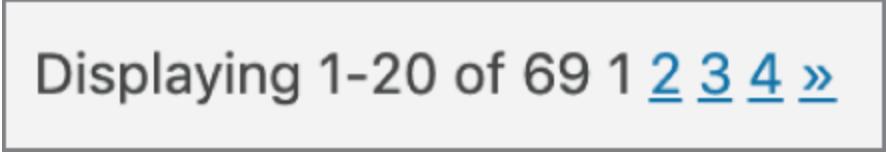
```

<div class="tablenav">
    <div class="tablenav-pages">
        <span class="displaying-num">Displaying 1-20 of
69</span>
        <span class="page-numbers current">1</span>
        <a href="#" class="page-numbers">2</a>
        <a href="#" class="page-numbers">3</a>
        <a href="#" class="page-numbers">4</a>
        <a href="#" class="next page-numbers">&raquo;</a>
    </div>
</div>

```

First, you need to wrap your pagination links in the `tablenav` and `tablenav-pages` div classes. The `displaying-num` class styles the records you view. The `page-numbers` class styles the page links in the familiar

WordPress format. Adding current or next to the link class can add some unique styles to those elements, as shown in [Figure 3-15](#).



Displaying 1-20 of 69 1 2 3 4 »

[FIGURE 3-15](#): Pagination style

Keeping your plugin design consistent with the WordPress user interface can reduce your plugins' learning curve because users will feel comfortable with the design and styles used. This can also make your plugins' design future-proof. If the WordPress core styles change down the road, your plugins' design will also change to match the new user interface, and you won't need to edit a single line of code!

SUMMARY

This chapter covered many different methods for integrating your plugin in WordPress. You certainly won't use every method discussed in every plugin you develop, but it's essential to understand what's available for use in your plugin.

4

Security and Performance

WHAT'S IN THIS CHAPTER?

- Understanding what security is
- Preventing XSS, CSRF, and other attacks
- Checking user permissions
- Validating and sanitizing data
- Creating secure SQL queries
- Caching data for optimal speed
- Temporarily storing expiring options

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

At this point, you've probably written some code and built at least one working plugin. However, is your plugin secure? Is it vulnerable to attacks? Is it optimized so that a user's site doesn't get taken down when they suddenly get a traffic boost? There are many points where security and optimization could break down on a site. Your job as a plugin developer is to make sure the code you write doesn't have vulnerabilities and isn't a bottleneck with resource usage.

WordPress has all the tools you need in place. What you must learn is when to use a particular function or feature. This chapter will walk you through the methods you need to know when securing and optimizing your plugin.

SECURITY OVERVIEW

When building a secure plugin, you must take steps to protect against cross-site scripting (XSS), cross-site request forgery (CSRF), SQL injection, privilege escalation, and other vulnerabilities. Unless you're a security expert, this is enough to scare anyone away from programming. While these terms may seem a bit scary, they don't need to be if you follow the best practices laid out by WordPress.

Some of the best talent in the programming world is devoted to making sure WordPress stays secure and offers the necessary tools for plugin authors to keep their code secure. It's possible your plugin could be run on thousands or even millions of websites, so making sure that it is secure is the most important thing you can do for its users.

What Securing Your Plugin Is

Securing your plugin means properly dealing with vulnerabilities and making sure data that is input or output is reliable. You must prevent malicious attacks while also making sure that legitimate user input/output does not produce unexpected behavior.

What Securing Your Plugin Is Not

Securing your WordPress plugin code in WordPress is not a difficult or unachievable task that only security experts can do. WordPress provides nearly every function you'll ever need for making sure your plugins are secure.

USER PERMISSIONS

The first step toward writing a secure plugin is figuring out whether a user has permission to perform an action. Not checking user permission will open your plugin to a privilege escalation vulnerability, which would grant users without permission the ability to perform sensitive actions that they are allowed to do.

If you've ever tried to visit a WordPress admin page while logged in as a non-administrator, you may have seen a message letting you know that you don't have sufficient privileges, as shown in [Figure 4-1](#).

You need a higher level of permission.

Sorry, you are not allowed to list users.

FIGURE 4-1: Insufficient privileges

WordPress uses a function named `current_user_can()` to check whether the user has the correct capability for viewing any admin page. It may check permission multiple times on a page for outputting different action links or buttons. In plugin development, you'll likely use this function extensively whenever you need to check whether a user can or cannot do something.

How to Check `current_user_can()`

When using the `current_user_can()` function, you'll check whether a user has permission to move forward with performing a sensitive action or output an error message and die. The following example checks whether the current user has permission to install plugins. If they do not, the code bails out with an error message.

```
<?php
if ( ! current_user_can( 'install_plugins' ) ) {
    wp_die( 'You do not have sufficient privileges to
install plugins.' );
}
```

Often, you'll want to check permission before outputting a link or some other HTML so that a user doesn't see actions in the UI they're not allowed to perform. The next example checks whether a user can manage options before outputting a link to the General Settings screen in the WordPress admin:

```
<?php if ( current_user_can( 'manage_options' ) ) {
printf(
    '<a href="%s">Manage Settings</a>',
    esc_url( admin_url( 'options-general.php' ) )
```

```
    );
}
```

The `install_plugins` and `manage_options` strings in the previous two examples are called *capabilities* in the WordPress environment. Capabilities are a part of the user role system, which you'll learn more about in [Chapter 9](#). For the purposes of security, you merely need to know whether the current user has a specific capability that you need to check against.

Do Not Check Too Early

Before checking whether a user has the correct authorization, you must wait until the current user's data has been set up. The `current_user_can()` function relies on `wp_get_current_user()`, a pluggable function. Pluggable functions can be overwritten by plugins. Therefore, they are not loaded until all plugins have been loaded.

Therefore, you must wait until WordPress has initialized before checking permission. The `init` hook is the earliest point in the load process that is a safe bet for such checks or any hook executed later.

Imagine that you wanted to build a plugin that provided a link to the user management screen to the WordPress toolbar (also called the *admin bar*). The toolbar has hooks that run after WordPress has initialized, so you can check whether a user has permission to edit users before appending the link.

See how the following complete plugin hooks into `wp_before_admin_bar_render` and checks whether the current user has the `edit_users` capability before adding a link to the toolbar:

```
<?php
/**
 * Plugin Name: Users Toolbar Link
 * Plugin URI: http://example.com/
 * Description: Adds a toolbar link to the users admin
 * screen.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'wp_before_admin_bar_render', 'pdev_toolbar' );

function pdev_toolbar() {
```

```
global $wp_admin_bar;  
  
if ( current_user_can( 'edit_users' ) ) {  
  
    $wp_admin_bar->add_menu( [  
        'id'      => 'pdev-users',  
        'title'   => 'Users',  
        'href'    => esc_url( admin_url(  
            'users.php' ) )  
    ] );  
}  
}
```

NOTE *When building plugins, always make sure to use an appropriate hook (see [Chapter 5](#), “Hooks,” for more information on hooks), such as `init`, so that all of WordPress’ functions have been loaded. This ensures that you neither trigger an error nor check for specific data, such as user permissions, before everything has been initialized.*

NONCES

In the previous section of this chapter, you learned how to check whether a user has permission to do something. On the surface, this seems like it would be everything you need to make sure a user can take a particular action. However, that's not always the case. Nonces were created to validate whether a user intended to take an action.

Authority vs. Intention

If a user has permission to perform an action (i.e., their role is granted a capability), it means they have the authority to do so. Any time you're in your WordPress admin, you can click links to do all sorts of things, such as publish a post, edit a category, or manage options. Any time you click a link to take an action, WordPress will verify that you have the authority to do so by checking `current_user_can()`.

But hackers are craftier than that. What happens if someone knew the exact URL on your site to delete a post, for example? They don't have the

authority to perform an action because they don't have sufficient permission on your site, so the URL wouldn't work for them. However, what if they tricked you, an administrator with permission, into clicking a link for deleting a post? Or even a more destructive action, such as deleting a user account? You would have the authority to take these actions, but it wasn't your intention to do so. This is called a *cross-site forgery request*.

NOTE *You may think that you'd never fall for such a trick. However, people hide these types of links with URL shortening services that are unrecognizable. This is prevalent on social media where links might be accompanied by something like a funny meme. The methods are always evolving, so you must protect your plugin users from them.*

What Is a Nonce?

A *nonce*, or a *cryptographic nonce*, is the abbreviation for “number used once.” In WordPress, a nonce is a short and seemingly random string that looks similar to a password. This string represents the following:

- One user
- One action, such as delete, update, or save
- One object, such as a post, term, or user
- One time frame that lasts 24 hours

You've probably seen these nonce strings in URLs in your WordPress admin from time to time. For example, the link to delete a user with the ID of 100 might look something like <http://example.com/wp-admin/users.php?action=delete&user=100& wpnonce=f73684c03c>. The f73684c03c at the end of the URL is the nonce. It is valid for only one time frame (24 hours) for one user (you) to take a specific action (delete) on the specific object (user) with the ID of 100.

If another person managed to get this link, even if they had a user account, the link would be invalid. After 24 hours, the link will also be invalid for your user account, and a new nonce will be created for the given action. As

you can see in [Figure 4-2](#), WordPress will output an expired link message if the nonce is invalid.

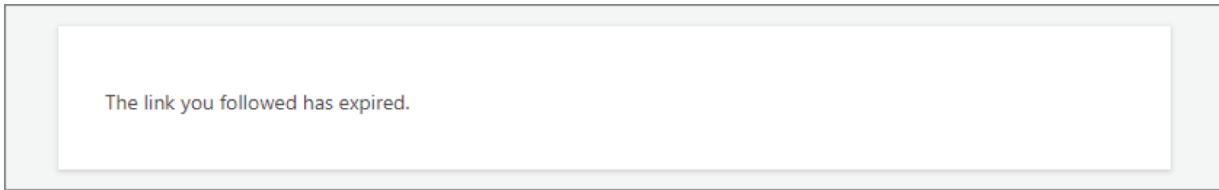


FIGURE 4-2: Expired link message

How to Create and Verify Nonces

WordPress has two methods of adding nonces. One method is for attaching a nonce to a URL. The second method is for outputting a hidden nonce field in HTML forms.

Creating a Nonce URL

To create a nonce URL such as the delete users URL (<http://example.com/wp-admin/users.php?action=delete&user=100&wpnonce=f73684c03c>), you would use the core `wp_nonce:url()` function.

```
<?php
wp_nonce_url( $url, $action = -1, $name = '_wpnonce' );
```

The first parameter of `$url` is the URL that you want to add a nonce to. The second parameter of `$action` is the string in which you build the nonce for one object and one action. The final parameter is a custom name for the nonce.

To build a delete-user URL, your code might look like the following:

```
<?php
$url = add_query_arg(
    [
        'action' => 'delete',
        'user'   => 100
    ],
    admin_url( 'users.php' )
);

return wp_nonce_url( $url, 'pdev_delete_user_' . 100 );
```

```
// Returns (with unique nonce):  
// http://example.com/wp-admin/users.php?  
action=delete&user=100&_wpnonce=f73684c03c
```

Of course, this URL doesn't delete a user account by merely visiting it. The nonce would need to be verified and code written to actually delete the user.

Creating a Nonce Field

When building forms, you should always add a nonce field to verify the intention of the user submitting the form. WordPress provides the `wp_nonce_field()` for handling this.

Consider the following form that allows a user to enter their name. The first form field added is the nonce field, which is a hidden field and not shown to the user. The field can go anywhere within the `<form>` element. However, it's good practice to add it prior to adding any other fields so that you don't forget to add it later.

```
<form method="post" action="">  
    <?php wp_nonce_field( 'pdev_nonce_action',  
    'pdev_nonce_name' ); ?>  
  
    <p>  
        <label>  
            Enter your name:  
            <input type="text"  
name="pdev_nonce_example" value="<?php echo  
                esc_attr( $value ); ?>"/>  
        </label>  
    </p>  
  
    <?php submit_button( 'Submit', 'primary' ); ?>  
</form>
```

Creating and Verifying a Nonce in a Plugin

Now take a look at how you would put all of this together to create a full plugin. The following code creates an admin page named "Nonce Example" with a form for entering a name. It adds a nonce field and verifies it upon submission of the form. If successfully verified, the name field value is saved in the database.

```
<?php
/**
 * Plugin Name: Nonce Example
 * Plugin URI:  http://example.com/
 * Description: Displays an example nonce field and verifies
 * it.
 * Author:      WROX
 * Author URI:  http://wrox.com
 */

add_action( 'admin_menu', 'pdev_nonce_example_menu' );
add_action( 'admin_init', 'pdev_nonce_example_verify' );

function pdev_nonce_example_menu() {

    add_menu_page(
        'Nonce Example',
        'Nonce Example',
        'manage_options',
        'pdev-nonce-example',
        'pdev_nonce_example_template'
    );
}

function pdev_nonce_example_verify() {

    // Bail if no nonce field.
    if ( ! isset( $_POST['pdev_nonce_name'] ) ) {
        return;
    }

    // Display error and die if not verified.
    if ( ! wp_verify_nonce( $_POST['pdev_nonce_name'],
        'pdev_nonce_action' ) ) {
        wp_die( 'Your nonce could not be verified.' );
    }

    // Sanitize and update the option if it's set.
    if ( isset( $_POST['pdev_nonce_example'] ) ) {
        update_option(
            'pdev_nonce_example',
            wp_strip_all_tags(
                $_POST['pdev_nonce_example']
            );
    }
}

function pdev_nonce_example_template() { ?>
```

```

<div class="wrap">
    <h1 class="wp-heading-inline">Nonce
Example</h1>

    <?php $value = get_option( 'pdev_nonce_example'
); ?>

    <form method="post" action="">

        <?php wp_nonce_field( 'pdev_nonce_action',
(pdev_nonce_name' ); ?>

        <p>
            <label>
                Enter your name:
                <input type="text"
name="pdev_nonce_example"
$value ); ?>">
            </label>
        </p>

        <?php submit_button( 'Submit', 'primary' );
?>
    </form>
</div>
<?php }

```

DATA VALIDATION AND SANITIZATION

Repeat the following sentence aloud: *all unknown data is unsafe*. Now one more time for good measure: *all unknown data is unsafe*.

This may seem like a paranoid view, but in computer programming it's healthy to be suspicious of all data going in and going out. Even if you've cleared both the authority and intent checks discussed earlier in this chapter, you should still think from the mind-set that some third party has found a way around those checks and is giving your plugin malicious data.

In this section, you'll learn why filtering data is important, how to validate and sanitize data by type, and what WordPress functions to use to keep data secure.

The Need for Data Validation and Sanitization

To show the importance of validating and sanitizing data, review the following form code:

```
<?php $full_name = $_POST['full_name']; ?>

<form action="" method="POST">
    <label>
        Full Name:
        <input type="text" name="full_name" value="<?php
echo $full_name; ?>"/>
    </label>
    <input type="submit" value="Submit"/>
</form>
```

This is a basic form that allows a user to enter their full name. However, there are two issues with the data in this case.

- The `$_POST` data in a real-world scenario would likely go to the database, so it must be sanitized to make sure that it's safe.
- The `$full_name` variable on output as the `value` attribute must be escaped to make sure it doesn't break out of the HTML and allow something like a malicious script to run.

To illustrate the problems with this form, input the following examples and check the results:

1. Nonmalicious example: John “Average” Doe
2. Malicious input: John"/>Password:<input name="password" value="1234"
3. Malicious input: John“<script>alert('XSS');</script>

Input 1 is an example of basic user input. Users input all sorts of characters both intentionally and unintentionally. It's your job to make sure this doesn't break things. The quotation marks in this case break the HTML in the form. They need to be encoded to their appropriate HTML entities.

Input 2 does the same thing as the first example. However, there is obvious malicious intent. Once the form is submitted, it breaks out of the HTML

and creates an additional password field. The hacker here is potentially attempting to change a password if they get lucky. The extra HTML should've been stripped when the form was posted.

Input 3 is an example of cross-site scripting. This is a serious vulnerability where hackers attempt to inject JavaScript to do bad things. For example, they may attempt to grab session or user cookies. The end goal is typically to gain access to privileged content or user accounts. If they're successful in gaining access, there's no limit to what they can do, such as taking down an entire site or adding hidden spam.

[Figure 4-3](#) illustrates the third example. As you can see, the form allows an arbitrary JavaScript code snippet to run. While an alert box is just an example, a malicious party could execute anything possible with JavaScript as if they had access to your site.



[FIGURE 4-3](#): Rogue JavaScript running

Now you can see why it's important to protect something as simple as a basic form on a web page from nonmalicious users with good intent as well as bad actors looking for ways to crack into a site. Remember that suspicion is healthy when dealing with untrusted data.

NOTE *On output, you must sanitize or “escape” data before sending it to the user's browser or screen. On input, you must make sure data is valid (validate) or filter it so that it becomes valid (sanitize).*

Good Practice: Identifying Potentially Tainted Data

Imagine that you created a survey to find out people's favorite fruit. Maybe you want to cross-reference that by age so that a client can better serve their customers at their online fruit shop. Your form may have a text input for visitors to enter their age and a drop-down for selecting from a predefined list of fruit. Consider the following code snippet that accepts the data that's input:

```
<?php
$safe = [];

$safe['age'] = absint( $_POST['age'] );

$valid_fruit = [
    'banana',
    'kiwi',
    'watermelon'
];

$safe['fruit'] = 'watermelon';

if ( in_array( wp_unslash( $_POST['fruit'] ), $valid_fruit,
true ) ) {
    $safe['fruit'] = wp_unslash( $_POST['fruit'] );
}
```

Notice that the first bit of code creates an empty array named `$safe`. It's good practice to create a separate variable early so that you can store data that has been validated or sanitized. Because you named it `$safe`, you'll recognize that it's the data that you'll want to send to the database or elsewhere after the form data has been processed.

Making sure that the age input is safe is relatively simple using WordPress' built-in `absint()` function, which makes sure that the data is an absolute integer. If it cannot be validated as an absolute integer, `0` will be returned.

The code example also only allows fruits to input from a specific group of allowed fruits. If an invalid value is input, the “safe” value is set to a default value. This is called *whitelist validation*. While not always possible, whitelisting is the method you should always strive to use to validate data because it rules out any nonvalid input. You should rarely, if ever, use blacklist validation, which only disallows specific values.

NOTE WordPress “magic quotes” PHP input super-globals, such as `$_POST` and `$_GET`. It’s good practice to remove slashes via the `wp_unslash()` function unless you’re using a sanitization function that also removes slashes.

Validating or Sanitizing Input?

Deciding whether to validate or sanitize input data is sometimes a tough choice. Ultimately, it comes down to a design decision on your part. After building plugins over time, you’ll start to get a feel for what the best choice is for types of inputs. Some types are easier than others.

Discussed in the previous section, whitelist validation is the ideal way to handle data input. This method often works best with form inputs that don’t allow the user to manually input data and have a limited set of options. Select drop-downs, radio inputs, and checkboxes are perfect for this type of validation because the allowed input should come only from your predefined options. Whitelist validation is a no-brainer in these cases.

Form fields that allow direct user input, such as text inputs or textareas, are where sanitization more often comes into play. Earlier in this chapter, you saw one method of handling an age input. The expected input is an integer. Invalid input becomes a 0. You may decide to not allow the form to be submitted unless it’s a valid age. For example, 1000 is also a valid integer. You should probably be a little skeptical of someone claiming to be the longest-living human ever. You may decide to validate that against an age range if it’s important to your plugin. Or, you may decide that you’ll allow any integer if the person’s age isn’t super important.

Imagine building a plugin with a user bio textarea. Validation probably isn’t ideal in this scenario. There’s more room for user error and no predefined options to choose from. Sanitization is your best bet. You’ll want to use a function such as `wp_kses()` to allow a limited set of HTML or use `wp_filter_nohtml_kses()` to strip all HTML.

There are some specific questions you should consider when deciding between validating and sanitizing.

- Will it be an inconvenience to the user if you reject all invalid data and require they resubmit the form after making corrections (e.g., removing invalid characters from an otherwise valid integer for an age input)?
- Can you make some assumptions about what the data should be if it's invalid and simply sanitize it (e.g., stripping HTML should affect a user's bio text input)?
- What will you do with the data immediately upon submission (e.g., attempt to send an email to an invalid email address)?

Validating and Sanitizing Cookbook

In this section, you'll learn about the many functions that WordPress provides for validating and sanitizing data. PHP also provides methods for filtering input data that you should study at <https://www.php.net/filter>.

Integers

For many cases, you can use PHP's built-in `intval()` to sanitize an integer or `is_int()` to determine whether a value is an integer. The following example shows both validating an integer value and sanitizing it:

```
<?php
$value = 100;

// Validate the integer.
return intval( $value ) === 100;

// Sanitize the integer.
return intval( $value );
```

Sometimes you may need to make sure a value is a positive integer. PHP provides the `abs()` function for that. However, WordPress provides a wrapper function named `absint()` that combines `intval()` and `abs()`. See how the following example takes the input of a negative integer and returns the absolute value of the number:

```
<?php
$value = -100;
```

```
// Returns 100.  
return absint( $value );
```

Generally, integers are easy to handle with these basic functions. However, when dealing with large integers, you may have to consider that computer systems have a maximum signed integer range. On 32-bit systems, that range is -2147483648 to 2147483647 . On 64-bit systems, the range is -9223372036854775807 to 9223372036854775807 . Any integer outside that range will be set to the maximum possible.

The best way to deal with integers outside these ranges is to treat them as strings instead. You could use PHP's `preg_replace()` to remove anything that's not a number, as shown in the following example:

```
<?php  
$value = '100000000000000000000000';  
  
// Strip anything that's not a digit.  
return preg_replace( '/\D/', '', $value );
```

Another option is to use `ctype_digit()`, which is preferable to regular expressions. However, PHP's `ctype` functions may be disabled on some installations, so you need to have a fallback plan in place if you don't know whether it's enabled.

```
<?php  
$value = '100000000000000000000000';  
  
// Strip anything that's not a digit.  
return ctype_digit( $value );
```

Arbitrary Text Strings

At times, you may have fields that allow a user to input arbitrary text strings, such as a text input or search field. WordPress provides the `sanitize_text_field()` function for this scenario, which will check for invalid UTF-8 characters, convert `<` characters to their appropriate HTML entities, strip all HTML tags, and strip octets. It will also remove line breaks, tabs, and extra whitespace.

Consider the following example of user input:

```
<?php
$value = "Jane Doe is      a <em>super cool</em> person!\n";
return sanitize_text_field( $value );
// Returns:
// "Jane Doe is a super cool person!"
```

As you can see, WordPress' `sanitize_text_field()` function removes all unwanted characters for you and leaves you with a nicely formatted text string.

There may be times when you want to strip all HTML tags but preserve other characters or text formatting. For those cases, you'll want to use WordPress' built-in `wp_strip_all_tags()` function as shown in the following snippet. Notice how the line break (`\n`) is preserved, but the HTML is removed.

```
<?php
$value = "John Doe \n is my <strong>best</strong> friend.
<script>alert( 'hello'
 );</script>";

return wp_strip_all_tags( $value );
// Returns:
// "John Doe \n is my best friend."
```

You may be thinking that PHP has its own built-in `strip_tags()` function, and you would be right. However, `strip_tags()` cannot properly handle more complex markup. Using the same input from the previous snippet, see how everything between the `<script>` tags is not removed.

```
<?php
$value = "John Doe \n is my <strong>best</strong> friend.
<script>alert( 'hello'
 );</script>";

return strip_tags( $value );
// Returns:
// "John Doe \n is my best friend.alert( 'hello' );"
```

The core WordPress `wp_strip_all_tags()` function should almost always be used in lieu of PHP's `strip_tags()` because this result is rarely what

you'll want.

Key and Identifier Strings

At times, you'll need to validate or sanitize internal identifier strings, or what are sometimes called *keys*. Keys are strings that can contain only alphanumeric characters, underscores, or hyphens. WordPress has a function named `sanitize_key()` that converts all letters to lowercase and strips any unwanted characters. However, if you need to validate instead of sanitize, you'll need to use PHP's `preg_match()`.

```
<?php
$value = 'pdev_100';

// Validate:
return preg_match( '/^a-z0-9-_]+$/i', '', $value );

// Sanitize:
return sanitize_key( $value );
```

In the example code, the validation check merely tests if the key is valid, returning `true` or `false`. It does this by using a regular expression. Anything not matching a lowercase letter, number, underscore, or hyphen will cause the check to fail and return `false`. You can learn more about using regular expressions in PHP at <https://www.php.net/manual/en/book.pcre.php>.

Email Strings

When allowing users to input an email address, WordPress provides the `is_email()` function for validating and provides `sanitize_email()` for sanitizing the email address.

```
<?php
$email = 'wrox@example.com';

// Validate.
return is_email( $email );

// Sanitize.
return sanitize_email( $email );
```

It's important to note that neither of these functions checks whether the email address is real. They check only that the string follows a specific pattern. The only way to verify the email address is to send an email and have the recipient perform an action, such as click a verification link that returns them to the site.

Emails such as `wrox@localhost`, a format used in some local networks without a TLD, would be an invalid address. If building a plugin under such a system, you would need to create a check for known characters, such as `@localhost`, or validate against a list of known email addresses if possible.

URLs

URLs need to be handled differently based on whether you're outputting to the browser, saving to the database, or redirecting a user to another URL.

URLs in HTML

The most common use case for URLs is to output them to the screen. They need to be escaped prior to output using WordPress' `esc_url()` function. This function removes some invalid characters and converts & characters to their HTML entity.

```
<?php $url = 'http://example.com'; ?>  
<a href="<?php echo esc_url( $url ); ?>">Example</a>
```

The `esc_url()` function doesn't make sure a link exists. Its purpose is to make sure that the value passed in doesn't break HTML and is safe to use as an attribute.

URLs in a Database

When storing a URL in the database, such as via a plugin setting, you don't want ampersands or other characters converted to their HTML entities. Instead, you want to make sure they're sanitized and safe for storing in the database. WordPress provides the `esc_url_raw()` function for this task.

```
<?php  
$url = 'http://example.com';  
  
update_option( 'pdev_url_setting', esc_url_raw( $url ) );
```

As you can see in the preceding code snippet, the `$url` value is sanitized before being saved to the database. You can learn more about saving options in [Chapter 3](#), “Dashboard and Settings.”

URLs in Redirects

For redirecting a user to another URL, WordPress provides the `wp_redirect()` function. This function both sanitizes the redirect URL and handles the redirect.

```
<?php
wp_redirect( 'http://example.com' );
exit;
```

`wp_redirect()` is vulnerable to open redirects if the URL is unknown, such as a URL set by user input. The function does not validate whether the URL matches the current host. For that reason, it's always better to use `wp_safe_redirect()` when the intention is for the user to be redirected to a page on the current site.

```
<?php
wp_safe_redirect( admin_url( 'options-general.php' ) );
exit;
```

NOTE *The `wp_redirect()` function does not exit automatically, so it is good practice to manually call `exit`; after performing a redirect.*

Escaping HTML and Attributes

When outputting content to the browser in which you want to disallow HTML, you can use WordPress' `esc_html()` function. Unlike `wp_strip_all_tags()` discussed earlier in this chapter, `esc_html()` will not remove the markup. Instead, it will convert `<`, `>`, and other characters to their character entities, as shown in the following example:

```
<?php
$value = '<h2>Hello, world!</h2>';
echo esc_html( $value );
```

```
// Returns:  
// <h2>Hello, world!</h2>
```

Likewise, the `esc_attr()` function escapes unwanted characters that you don't want to break HTML when used as an attribute value. The following example shows how to make sure a variable is safe to use as an HTML element class:

```
<?php $class = 'example'; ?>  
  
<p class="<?php echo esc_attr( $class ); ?>">  
    The quick brown fox...  
</p>
```

HTML

At times, you may need to sanitize HTML to make sure that it's safe for input into the database or output on the front end. Often, this is when accepting data passed from a `<textarea>` field. Good examples of this are the WordPress post editor or its comment form on the front end.

Forcing Balanced Tags

The first function in your arsenal will be the core `force_balance_tags()` function. This function does not validate tags. However, it adds missing tags, as shown in the following code snippet:

```
<?php  
$html = '<p>I have a missing closing tag!';  
  
return force_balance_tags( $html );  
  
// Returns:  
// '<p>I have a missing closing tag!</p>';
```

The function will also properly balance tags that are out of order. Consider the following example where the `` and `` tags are mismatched:

```
<?php  
$html = '<p>My superhero name is <strong><em>Super  
Jane</strong></em></p>';  
  
return force_balance_tags( $html );  
  
// Returns:
```

```
// '<p>My superhero name is <strong><em>Super Jane</em></strong></p>';
```

Sanitizing HTML

WordPress includes a PHP script called KSES, which is a reverse acronym that stands for “KSES Strips Evil Scripts.” The script is useful for filtering untrusted HTML on input and output. There are multiple KSES functions. However, the primary function that you’ll use is `wp_kses()`, which allows developers to pass in an array of allowed HTML elements and allowed attributes for each element.

Imagine you had an HTML string you wanted to return for output but only wanted to allow `` and `` tags. `wp_kses()` allows you to limit the string, as shown in the following example:

```
<?php
$allowed = [
    'strong' => [],
    'em'      => []
];

$html = '<h1>A <strong>Bold</strong> and <em>Italic</em> Header</h1>';

return wp_kses( $html, $allowed );

// Returns:
// A <strong>Bold</strong> and <em>Italic</em> Header
```

As you can see, the `<h1>` tag in the original `$html` string was stripped because it was not defined in the `$allowed` array of elements. You can take this further by also defining the allowed attributes for any allowed elements. See how the following `$allowed` array allows specific attributes:

```
<?php
$allowed = [
    'strong' => [
        'class' => []
    ],
    'em'      => [
        'class' => []
    ],
    'a'       => [
        'href'  => []
    ],
];
```

```

        'title' => [],
        'class' => []
    ]
];

```

As you may guess, defining a huge list of elements and attributes can become unwieldy if you plan to allow many elements. `wp_kses()` is usually best when you have a specific subset of HTML that you want to allow. If you want to allow for almost any valid HTML but don't want to define the elements, it's usually better to use the `wp_kses_data()` function provided by WordPress, which only allows tags defined in WordPress' allowed tags list.

Consider the following input where a malicious user passes in a `<script>` tag to run JavaScript. KSES will clean that up and make it safe for input or output.

```

<?php
$html = '<p>Hello, world! My name is <strong>John
Doe</strong>.
    I want to insert <script>alert( "XSS" );</script>';

return wp_kses_data( $html );

// Returns:
// Hello, world! My name is <strong>John Doe</strong>. I
// want to insert
// alert( "XSS" );

```

JavaScript

WordPress provides the `esc_js()` function for escaping quotes and ampersands while correcting line endings. This function is not meant for making entire JavaScript code blocks safe. Its purpose is for escaping JavaScript used in HTML attributes. The following code snippet shows a button that when clicked will create an alert box with the text that has been made safe for JavaScript:

```

<?php $value = 'Hello'; ?>

<button onclick="alert( '<?php echo esc_js( $value ); ?>' );
">Click Me</button>

```

The usefulness of `esc_js()` is limited. Often, it's better to use `wp_json_encode()` combined with `esc_attr()` on output. A more likely

scenario is that you'll need to pass custom data to a data attribute. Then, later use that in a custom script. Note how the following snippet adds a hex color code to the data-pdev attribute:

```
<?php $data = [
    [
        'color' => '#000'
    ]
]; ?>

<div data-pdev="<?php echo esc_attr( wp_json_encode( $data ) );
); ?>">
</div>
```

That code would result in the following HTML output:

```
<div data-pdev="["color":"#000"]">
</div>
```

You should avoid outputting JavaScript directly in HTML if possible, even if you have PHP variables that need to be processed and filtered first. Instead, you should opt to use `wp_localize_script()` to make data available to your JavaScript files, which you'll learn about in [Chapter 6](#).

Environment and Server Variables

In PHP, you cannot trust the `$_SERVER` array and must treat it as if it were any other unknown variable. Imagine that you wanted to display a welcome message based on where the current site visitor came from. You'd use the `HTTP_REFERER` key to get that data.

```
<?php if ( isset( $_SERVER['HTTP_REFERER'] ) ) : ?>
    Welcome, visitor from <?php echo
    $_SERVER['HTTP_REFERER']; ?>.
<?php endif; ?>
```

The preceding code is unsafe. `$_SERVER['HTTP_REFERER']` can easily be spoofed and used to execute a malicious script. Because you'd expect a URL in this scenario, you'd combine it with the `esc_url()` function.

```
<?php if ( isset( $_SERVER['HTTP_REFERER'] ) ) : ?>
    Welcome, visitor from <?php echo esc_url(
    $_SERVER['HTTP_REFERER']); ?>.
<?php endif; ?>
```

Server variables can be as dangerous as any other type of unknown data. Always assume anything coming from `$_SERVER`, `$_COOKIE`, `$_GET`, `$_POST`, or any other super-global is malicious. Then, validate or sanitize based on what type of data you expect.

Arrays of Data

At times, you'll need to sanitize an array of the same type of data. Instead of manually typing out a line of code to filter each item in the array or running a complicated loop, use PHP's `array_map()` function to map a sanitization function to each array value.

For example, the following code has an array of HTML classes that need to be sanitized for a `<figure>` element that wraps an image. You only need to pass the `sanitize_html_class()` function and the array of classes to `array_map()` to handle this for you.

```
<?php
$classes = [
    'media',
    'media-object',
    'media-image'
];

$classes = array_map( 'sanitize_html_class', $classes );
?>

<figure class="<?php echo esc_attr( join( ' ', $classes ) ); ?>">
    
</figure>
```

Of course, you'll also want to escape the final output with the `esc_attr()` function covered earlier in this chapter, which cleans up the final output.

```
<figure class="media media-object media-image">
    
</figure>
```

Nearly any sanitization or validation function can be combined with `array_map()`. You can also use a custom callback for more complex data. This function will be one of the most useful tools in the toolbox as you start building larger plugins.

Database Queries

Database queries are an important area to secure against SQL injection attacks. The following example shows a query for handling a user login, which is insecure. Take note of the `$password` variable.

```
<?php
$login      = 'hacker';
$password  = "123456' OR 1='1";

return "SELECT * FROM users WHERE `login` = '$login' AND
`pass` = '$password'";

// Returns:
// SELECT * FROM users WHERE `login` = 'hacker' AND `pass` =
'123456' OR 1='1'
```

Because the variables are not escaped, the `$password` variable successfully exploits a vulnerability by closing out a quote mark and adding `OR 1='1'`, which is always true. This would allow a malicious actor to successfully log into the site without having appropriate credentials.

To prevent this, you should use WordPress' `esc_sql()` function to escape any variables passed into a SQL query. The following example shows how using this function ensures the quote marks from `$password` are escaped and cannot be exploited:

```
<?php
$login      = esc_sql( 'hacker' );
$password  = esc_sql( "123456' OR 1='1" );

$sql = "SELECT * FROM users WHERE `login` = '$login' AND
`pass` = '$password'";

// Returns:
// SELECT * FROM users WHERE `login` = 'hacker' AND `pass` =
'123456\' OR 1=\\'1'
```

NOTE `esc_sql()` is meant for escaping strings that will be used in a query. Any strings should always be wrapped in quotes. This function shouldn't be used for numbers, fields, or other unquoted data.

FORMATTING SQL STATEMENTS

WordPress provides API functions for nearly everything you'll ever need to do when getting, setting, updating, or deleting data in the database. You'll learn about most of these functions throughout this book. However, there are rare cases where you'll need to interact more directly with the database, and WordPress provides the tools necessary for handling this too.

The `$wpdb` Object

When interacting directly with the database, WordPress includes a `wpdb` class. On every page load, WordPress stores a single object instance of this class in the global `$wpdb` variable.

You should always use the `$wpdb` variable for database interactions instead of using PHP functions such as `mysql_query()` for two reasons.

- WordPress provides additional security functionality to protect against SQL injection attacks.
- Users may decide to use a different database engine such as PostgresSQL, so `mysql_*`() functions may not work.

You can use the `$wpdb` object to access any of the tables in the database created by WordPress on installation. You can also access tables created by your plugin or tables from third-party plugins.

NOTE *When accessing `$wpdb`, make sure to globalize the variable within functions and classes. Otherwise, the variable will be out of scope and not available.*

Why `wpdb` Methods Are Superior

The `$wpdb` object allows you to read, insert, update, or delete data from database tables. The following is an example of using the PHP `mysql_*`() functions for connecting to a database and updating a post title:

```
<?php
$title = esc_sql( 'New Post Title' );
```

```

$id      = absint( 100 );

mysql_connect( DB_HOST, DB_USER, DB_PASSWORD )
    OR die( 'Could not connect: ' . mysql_error() );

mysql_select_db( DB_NAME );

mysql_query( "UPDATE wp_posts SET post_title = '$title'
WHERE ID = $id" );

```

There are two primary issues with this code. First, it is a lot of code to write for a simple database query. Writing more code allows for more potential mistakes. Second, the `wp_posts` table is hard-coded, so it may not be correct. WordPress allows users to change the default `wp_` prefix for table names to something custom. Compare the preceding code with the WordPress method of handling the same query, shown here:

```

<?php
$title = esc_sql( 'New Post Title' );
$id    = absint( 100 );

$wpdb->query( "UPDATE $wpdb->posts SET post_title = '$title'
WHERE ID = $id" );

```

This is much shorter in length and easier to read. You can also see that `$wpdb->posts` points to the correct table for posts. If the user chose to use `pdev_` as their table prefix, this would become `pdev_posts`.

You can even take this one step further and directly update an entire row in a table via the `$wpdb->update()` method. The following example performs the same action of updating a post's title for a given post ID:

```

<?php
$data  = [ 'post_title' => $title ];
$where = [ 'ID' => $id ];

$wpdb->update( $wpdb->posts, $data, $where );

```

You can also update multiple columns at once with this method by adding the keys and values to the `$data` array. This will save you from writing out longer queries.

All-in-One Methods

WordPress provides methods for interacting with an entire table row that are easier to use than memorizing SQL syntax and also handle sanitizing the data for you. In the previous section, you learned about the `update()` method, but WordPress also has an `insert()` method.

`$wpdb->update()`

When you want to update an existing row in a particular table, you should use the `$wpdb->update()` method, which accepts five parameters.

- `$table` (required): The database table name to perform the update on.
- `$data` (required): An array of columns (keys) to update with the new values to insert into the row. The columns and values should be raw values and not SQL-escaped.
- `$where` (required): An array of where clauses in the form of column and value pairs.
- `$format` (optional): An array of formats that get mapped to the values in the `$data` array. Use `%s` for strings, `%d` for integers, and `%f` for floats. If the parameter is a string, it will be used to format all the values.
- `$where_format` (optional): An array of formats that get mapped to the `$where` array. It works the same as the `$format` parameter.

Building from the previous example of updating the post title, imagine that you also wanted to change the post author. WordPress stores the post title as a string and the post author as an integer (user ID). The following code snippet shows how to format the code to update the post in the database:

```
<?php
$data  = [
    'post_title'  => 'New Post Title',
    'post_author' => 42
];
$where = [
    'ID' => $id
];
$format = [
    '%s', // Maps to post_title
```

```

        '%d' // Maps to post_author
    ];

$where_format = [
    '%d' // Maps to ID
];

return $wpdb->update( $wpdb->posts, $data, $where, $format,
$where_format );

// Returns number of updated columns on success and false on
failure.

```

As you can see, this method provides a structured method for updating rows in a database table without having to write any SQL syntax or manually sanitize the data.

\$wpdb->insert()

To insert a new row into a table, you should use the `$wpdb->insert()` method, which accepts three parameters.

- `$table` (required): The database table name to perform the update on.
- `$data` (required): An array of columns (keys) to update with the new values to insert into the row. The columns and values should be raw values and not SQL-escaped.
- `$format` (optional): An array of formats that get mapped to the values in the `$data` array. Use `%s` for strings, `%d` for integers, and `%f` for floats. If the parameter is a string, it will be used to format all the values.

Assume you wanted to insert a new row into a custom table that had a column named `column_1` that accepts a string input and `column_2` that accepts an integer input. You could easily do this with `$wpdb->insert()`.

```

<?php
$data = [
    'column_1' => 'A text string',
    'column_2' => 100
];

$format = [
    '%s',

```

```
        '%d'  
];  
  
return $wpdb->insert( $wpdb->custom, $data, $format );  
  
// Returns number of updated columns on success and false on  
failure.
```

NOTE *Do not hard-code the wp_ prefix when referencing a core WordPress or custom database table. Always target \$wpdb->table_name or use \$wpdb->prefix to get the prefix.*

Common Methods

Sometimes, database queries will be more complex than a basic UPDATE or INSERT. You'll need to write some SQL syntax and use one of the core methods for interacting with the database.

Select a Variable

The \$wpdb->get_var() method will return a single variable from the database. If no data is found, it will return null. The following example shows how to grab the number of published posts on a WordPress installation:

```
<?php  
$query = "SELECT COUNT(ID) FROM {$wpdb->posts} WHERE  
post_status = 'publish"';  
  
return $wpdb->get_var( $query );  
  
// Returns the number of published posts.
```

Select a Row

To select an entire row from the database, use the \$wpdb->get_row() method. It allows you to return the row's data as an object (default), associative array, or numeric array. If no value is found, it will return null.

```
<?php  
$wpdb->get_row( string|null $query = null, string $output =  
OBJECT, int $y = 0 );
```

This function accepts three parameters:

- `$query` (optional): The SQL query.
- `$output` (optional): The return type of the method. Acceptable values are `OBJECT` (returns an object), `ARRAY_A` (returns an associative array), and `ARRAY_N` (returns a numeric array).
- `$y` (optional): The row to return. Note that rows are indexed from 0.

Imagine you needed to grab the admin user and all of the user's data. You could do so with the following snippet:

```
<?php
$query = "SELECT * FROM {$wpdb->users} WHERE user_login =
'admin';

return $wpdb->get_row( $query );
```

That code will return an object with each of the admin user's columns from the database.

```
object(stdClass) (10) {
    ["ID"] => string(1) "1"
    ["user_login"] => string(5) "admin"
    ["user_pass"] => string(34) "xxxxxxxx"
    ["user_nicename"] => string(5) "admin"
    ["user_email"] => string(23) "example@example.com"
    ["user_url"] => string(24) "http://example.com"
    ["user_registered"] => string(19) "2019-09-24
20:22:42"
    ["user_activation_key"] => string(0) ""
    ["user_status"] => string(1) "0"
    ["display_name"] => string(14) "John Doe"
}
```

Select a Column

To select an entire column or part of a column from a table, you should use the `$wpdb->get_col()` method.

```
<?php
$wpdb->get_col( string|null $query = null, int $x = 0 );
```

This function accepts two parameters.

- `$query` (optional): The SQL query.
- `$x` (optional): The column to return. Note that columns are indexed from 0.

Assume you needed to grab the email address for every user for the WordPress installation. The following will return each of the emails as a numeric array:

```
<?php
$query = "SELECT user_email FROM {$wpdb->users}";
return $wpdb->get_col( $query );
```

The returned data would look similar to the following:

```
[  
    0 => 'email_a@example.com',  
    1 => 'email_b@example.com',  
    2 => 'email_c@example.com'  
]
```

You could take that a step further and use those addresses to fire off an email to every user. The following example lets users know the site will be offline next Monday:

```
<?php
$query = "SELECT user_email FROM {$wpdb->users}";
$emails = $wpdb->get_col( $query );
foreach ( $emails as $email ) {
    wp_mail(
        $email,
        'Site Offline',
        'The site will be offline next Monday.'
    );
}
```

Select Generic Results

To grab a set of generic results from multiple rows, use the `$wpdb->get_results()` method. It accepts a first parameter of a valid SQL query and an optional second return-type parameter of `OBJECT` (default), `ARRAY_A`, or `ARRAY_N`.

You could do something more complex such as get all the years on a WordPress installation and count the number of posts.

```
<?php
$query = "SELECT YEAR(post_date) AS `year`, count(ID) as
posts
        FROM $wpdb->posts
        WHERE post_type = 'post' AND post_status =
'publish'
        GROUP BY YEAR(post_date)
        ORDER BY post_date DESC";

return $wpdb->get_results( $query, ARRAY_A );
```

By using `ARRAY_A` for the return type, the preceding code will return an associative array similar to the following:

```
[

[0] => [
    "year" => "2019",
    "posts" => "32"
],
[1] => [
    "year" => "2018",
    "posts" => "1"
],
[2] => [
    "year" => "2017",
    "posts" => "3"
]
]
```

Generic Queries

WordPress also supports generic SQL queries via the `$wpdb->query()` method. The method returns an integer based on the number of rows affected or `false` if an error occurs.

Imagine you had a lot of spam users creating accounts with a common URL for the user website field. You could delete all of these users with one query.

```
<?php
$query = "DELETE from {$wpdb->users}
        WHERE user_url
        LIKE '%spam.example.com%'";
```

```
return $wpdb->query( $query );
```

Any type of query is possible with the `$wpdb->query()` method. You're not limited to doing a simple `SELECT` as you can see with the preceding usage of `DELETE`.

Protecting Queries against SQL Injections

Before you begin building custom queries in a plugin, it's important that you familiarize yourself with the `$wpdb->prepare()` method. Often, you'll be handling dynamic data rather than hard-coded example code. It's paramount that you protect your plugin's queries against an SQL injection attack.

The following example queries all the titles of the posts written by the user with an ID assigned to the `$user_id` variable. Because `$user_id` is unknown, and therefore untrusted, the query string needs to be sanitized for safety using the `$wpdb->prepare()` method before adding the query string to `$wpdb->get_results()`.

```
<?php
$user_id = 1;

$unsafe_query = "SELECT post_title
    FROM {$wpdb->posts}
    WHERE post_status = 'publish'
    AND post_author = %d";

$safe_query = $wpdb->prepare( $unsafe_query, $user_id );

return $wpdb->get_results( $safe_query );
```

When using `$wpdb->prepare()`, you use `%s` as a placeholder for strings and `%d` as a placeholder for integers. You don't need to add quotes around these placeholders.

SECURITY GOOD HABITS

Writing secure plugins is not about doing one thing right at one point in time. It's not something you set and forget. You must approach security at every point in the design of your plugin.

Humans are imperfect. You'll make a security mistake at some point in your plugin development career. All programmers do. It's not something to feel shame over. It's what you do when addressing security issues that matters, which means being proactive about getting a fix out to users to close the security hole as quickly and safely as possible.

The number-one takeaway from this chapter is that you should trust nothing and no one.

However, there's no need to be scared away from building plugins because of potential security problems. If you stay mindful of the common issues and utilize the functions outlined in this chapter, you'll be well on your way to building plugins professionally.

Follow these simple practices and form good habits of them:

- Try to break your own code. Test various situations, add odd characters to inputs, and throw nonstandard scenarios at anything you build.
- Begin thinking about security from the start of your plugin. Don't wait until you get a working prototype or enter the beta testing phase before checking for potential exploits. Make it a habit to attempt to write secure code from the outset.
- As outlined in [Chapter 2](#), “Plugin Framework,” document your code. Having good inline comments will help you understand whether a particular block of code is doing what it's supposed to do when you look at it later.
- Assume all unknown data is dangerous. This is worth repeating a few times.

PERFORMANCE OVERVIEW

WordPress provides two separate methods for caching data over a period of time.

- **Cache API:** By default, data persists only for a single page load, unless a user installs a caching plugin. This is useful when storing data from expensive operations.

- **Transient API:** This is a wrapper around the Options API discussed in [Chapter 3](#). However, the transients are stored based on a maximum expiration date.

In the following sections, you'll learn how to use each when building your plugins. Both have their benefits for storing data that you'll need to later access without having to perform heavy operations over and over.

CACHING

Caching is a way to store data for later use. WordPress doesn't provide a single method of persistently storing cached data. Instead, it provides the Cache API, which is a standardized set of functions that all plugins can use. Because there's a singular API to use, caching plugins have a standard method for working with data that should be cached. This way, users can determine the ideal caching method for their own setups, which can vary based on where their website is hosted.

While WordPress doesn't persistently cache data out of the box, it does cache data on each page load. Therefore, even if a user doesn't have a caching solution in place, you can set and retrieve data multiple times on a single page load without fear of taxing the database or server.

WordPress uses the Cache API internally for many operations, so you don't always necessarily have to interact with the API in your own code. However, when you have data that is expensive to generate, you should cache it so that it can be loaded quickly.

Saving Cached Data

WordPress provides three primary functions for storing cached data.

```
<?php
wp_cache_add( int|string $key, mixed $data, string $group =
'', int $expire = 0 );

wp_cache_replace( int|string $key, mixed $data, string
$group = '', int $expire = 0 );

wp_cache_set( int|string $key, mixed $data, string $group =
'', int $expire = 0 );
```

As you can see, each function looks similar. `wp_cache_add()` saves cached data. `wp_cache_replace()` replaces existing cached data. And `wp_cache_set()` is a combination of the previous two functions. In most situations, you'll use it as an all-in-one method simply because it handles both adding and replacing.

Each function accepts four parameters:

- `$key` (required): A unique ID to cache the data by. This key doesn't have to be unique across multiple data groups but must be unique within a group.
- `$data` (required): The data that you want to store and access later.
- `$group` (optional): A group ID to store multiple pieces of cached data.
- `$expire` (optional): How long the data should be stored in number of seconds. By default, this is set to 0, which tells caching systems to store the data indefinitely.

Imagine you wanted to build a plugin that gathered five posts related to the current blog post by category. The first thing you'd need to do is perform a query that retrieved the posts. You could store the result of this query using the `wp_cache_set()` helper.

```
<?php
$post_id      = get_the_ID();
$categories = get_the_category();

$posts = get_posts( [
    'category' => absint( $categories[0]->term_id ),
    'post__not_in' => [ $post_id ],
    'numberposts'  => 5
] );

if ( $posts ) {
    wp_cache_set( $post_id, $posts, 'pdev_related_posts',
    DAY_IN_SECONDS );
}
```

Loading and Using Cached Data

When you need to load cache data, WordPress provides the `wp_cache_get()` function. You can retrieve data by key (first parameter) and group (second parameter). If no data is found, the function will return `false`.

Using the related posts example from the previous section, check whether there are posts cached. If so, you can output them as a list.

```
<?php
$post_id = get_the_ID();

$post = wp_cache_get( $post_id, 'pdev_related_posts' );

if ( $post ) {
    echo '<ul>';

    foreach ( $post as $post ) {
        printf(
            '<li><a href="%s">%s</a></li>',
            esc_url( get_permalink( $post->ID ) ),
            esc_html( get_the_title( $post->ID ) )
        );
    }

    echo '</ul>';
}
```

Deleting Cached Data

When you need to delete cached data, use the WordPress `wp_cache_delete()` function. Like `wp_cache_get()`, it accepts a `$key` and a `$group` as the first and second parameters. The function will return `true` on success and `false` on failure.

Building from the related posts example, use the following code to delete the current post's cached related posts data:

```
<?php
$post_id = get_the_ID();

return wp_cache_delete( $post_id, 'pdev_related_posts' );
```

Caching Data within a Plugin

In this section, you'll build a simple related posts plugin using the functions described in the previous sections. The plugin will append a related posts list to the end of the post content on single post views. It will do so using a filter on a hook named `the_content`. You'll learn more about hooks in [Chapter 5](#).

There's a few things the plugin must do.

- Check whether the user is viewing the singular post.
- Attempt to get the cached related posts.
- If no cached data is found, query new posts and cache them.
- Append the related posts HTML to the post content.

```
<?php
/**
 * Plugin Name: Related Posts
 * Plugin URI: http://example.com/
 * Description: Displays a list of related posts on singular
views.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_filter( 'the_content', 'pdev_related_posts' );

function pdev_related_posts( $content ) {

    // Bail if not viewing a single post.
    if ( ! is_singular( 'post' ) || ! in_the_loop() ) {
        return $content;
    }

    // Get the current post ID.
    $post_id = get_the_ID();

    // Check for cached posts.
    $posts = wp_cache_get( $post_id, 'pdev_related_posts' );
    if ( ! $cache ) {
        $categories = get_the_category();
```

```

$posts = get_posts( [
    'category' => absint( $categories[0]-
>term_id ),
    'post__not_in' => [ $post_id ],
    'numberposts' => 5
] );

// Save the cached posts.
if ( $posts ) {
    wp_cache_set(
        $post_id,
        $posts,
        'pdev_related_posts',
        DAY_IN_SECONDS
    );
}

// If posts were found at this point.
if ( $posts ) {

    $content .= '<h3>Related Posts</h3>';

    $content .= '<ul>';

    foreach ( $posts as $post ) {
        $content .= sprintf(
            '<li><a href="%s">%s</a></li>',
            esc_url( get_permalink( $post->ID
) ),
            esc_html( get_the_title( $post-
>ID ) )
        );
    }

    $content .= '</ul>';
}

return $content;
}

```

The plugin will output a Related Posts list that looks similar to [Figure 4-4](#).

Related Posts

- [Block: Image](#)
- [Block: Cover](#)
- [Block: Gallery](#)
- [Block: Columns](#)
- [Block: Quote](#)

FIGURE 4-4: Related Posts list

TRANSIENTS

Sometimes you need to store temporary data in the database. For example, you may build a plugin that pulls your latest tweets from [Twitter.com](#) or fetches an RSS feed from a third-party site. You wouldn't want to hit those websites for each page request on your site. Instead, you'd store the data for a limited time.

The Transient API is similar to the Options API with an added expiration component. Unlike the Cache API discussed earlier in this chapter, it does not require the use of a caching plugin to persistently store the data. However, if a caching plugin is installed and activated on the user's site, the Transient API will revert to using the Cache API.

Saving an Expiring Option

Imagine that you're building a plugin that pulled in a video tutorial on how to use WordPress from a third-party site. The latest video is titled *How to Install WordPress*. You want to store the latest video for one day.

```
<?php
set_transient( 'pdev_video_tutorial', 'How to Install
WordPress', DAY_IN_SECONDS );
```

The first parameter is the unique ID for your transient. It should be prefixed like any other global identifier, as discussed in [Chapter 2](#). The second parameter is the latest video title. In reality, this can be any arbitrary data. The final parameter is the number of seconds to store the data.

DAY_IN_SECONDS is a constant defined by WordPress for easily getting a day in seconds.

Retrieving an Expiring Option

When retrieving an option stored as a transient, you will use the `get_transient()` function provided by WordPress. Use this to grab the latest tutorial video title.

```
<?php
$video = get_transient( 'pdev_video_tutorial' );
```

The `$video` variable will return one of two things.

- If the transient is stored in the database, it will return the value of the option.
- If retrieval fails or no value is stored, it will return `false`.

Deleting an Expiring Option

Like with setting and getting transients, WordPress packages a `delete_transient()` function for deleting a transient from the database. You may need to use this to invalidate a transient, such as when uninstalling the plugin. To remove the tutorial video transient, use the following code:

```
<?php
delete_transient( 'pdev_video_tutorial' );
```

Generally, you should rarely need to delete a transient. WordPress will automatically delete expiring transients from the database for you.

A Practical Example Using Transients

Now you can put together some basic plugin functions for setting and retrieving a tutorial video.

```
<?php
// Fetches video from third-party website.
function pdev_fetch_video_title() {
    // Connect to an API to fetch video.
    return $title;
}
```

```

// Returns the video title.
function pdev_get_video_title() {

    // Get transient.
    $title = get_transient( 'pdev_video_tutorial' );

    // If the transient doesn't exist or is expired,
    // refresh it.
    if ( ! $title ) {
        $title = pdev_fetch_video_title();

        set_transient( 'pdev_video_tutorial', $title,
        DAY_IN_SECONDS );
    }

    return $title;
}

```

You'd need to actually interact with a remote API and make sure the data you retrieve is secure to create a working set of code. You'll learn more about getting remote data via WordPress' HTTP API in [Chapter 12](#), “REST API.”

Technical Details

Because transients are volatile in nature, they benefit from caching plugins. There are various types of caching plugins that save data in different ways. Some cache systems store data in fast memory or in files on the server. While transients are stored in the database by default, you can never assume that's where they'll be when retrieving them. Always use the Transient API functions to ensure that you get the correct data.

Transient Ideas

Whenever you need to store data with a short lifetime, you should almost always use transients. The following list covers some examples of when you may want to use the Transient API:

- ▶ Displaying the latest dollar values for certain types of cryptocurrency such as Bitcoin or Ethereum on a cryptocurrency site
- ▶ Displaying the latest tweets from a Twitter account

- Showcasing the users who like a site's Facebook page
- Fetching an article from an RSS feed
- Pulling in a music playlist from a remote website

SUMMARY

In this chapter, you got an overview along with practical examples of securing data and optimizing data for performance. Security is far more important than optimization. An optimized website is useless if the data coming in or going out is not secure in the first place. Make security a priority from the first line of code you write and never trust unknown data. Then look for areas that will benefit from storing data via the Cache API or Transient API.

5

Hooks

WHAT'S IN THIS CHAPTER?

- Creating actions for action hooks
- Creating filters for filter hooks
- Using hooks with PHP classes and anonymous functions
- Creating custom hooks in plugins
- Finding hooks within WordPress

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

Hooks are the “bread and butter” of WordPress. They’re the essential feature you need to build plugins. Hooks allow plugins to *plug in* to WordPress and “do stuff.” While it is possible to build a plugin that doesn’t rely on the hooks system, such plugins would be a rarity. The majority of plugins will use at least one hook.

Once you start learning how hooks work, you will begin understanding why WordPress is such a powerful platform. It has allowed a community of thousands of developers to build upon it for years, which has in turn propelled the software’s growth beyond systems without any easy way for third-party developers to build upon them.

UNDERSTANDING HOOKS

In WordPress, *hooks* are the primary feature of the Plugin API. They enable plugin developers to “hook” into the WordPress flow and alter how it works without modifying the core code. By not altering core WordPress, users can

upgrade to newer versions without losing any modifications that their plugins have done.

Plugins would have no way to modify how WordPress works without hooks. In many ways, hooks represent what plugins are. The lessons you learn in this chapter are essential for utilizing the techniques that you'll learn in later chapters. Without hooks, the plugins you build would have no way to modify how WordPress functions.

WordPress has two primary types of hooks: action hooks and filter hooks. *Action hooks* allow you to execute a function (action) at a certain point in the WordPress flow. *Filter hooks* allow you to manipulate (filter) output passed through the hook.

NOTE *While hooks are the essential ingredient of the Plugin API, plugins aren't the only things that use the system. Themes can use hooks, and WordPress itself uses its own hooks. If you browse the core source code, you'll often see WordPress attaching actions or filters to hooks.*

ACTIONS

Action hooks enable you to execute code at a specific moment in the WordPress loading process. It's easiest to think of these as "events" that happen at certain stages. By attaching a function to the action hook, your plugin is telling WordPress it wants to do something at this event.

The `do_action()` function in WordPress creates and fires an action hook. When attaching an action to the hook, you wouldn't call this function directly. However, there may come a point where you will want to create custom action hooks for your plugin, which you will learn about in the "Creating Custom Hooks" section of this chapter.

```
<?php
do_action( $tag, $arg = '' );
```

The function accepts two parameters.

- `$tag`: The name or identifier for the action hook.

- `$arg`: Value(s) passed to registered actions. This is where things get tricky for first-time developers. This parameter may be split into any number of parameters, which will depend on the specific action hook.

The following is an example of what an action hook would look like with multiple parameters:

```
<?php
do_action( $tag, $arg_a, $arg_b, $arg_c );
```

One of the most common hooks in WordPress is the `wp_head` action hook. It is fired within the opening `<head>` and closing `</head>` tags on the frontend of the website. It's a useful hook for SEO plugins to add meta and Open Graph social media tags. Take a look at this action hook:

```
<?php
do_action( 'wp_head' );
```

As you can see, this particular action hook passes no parameters. Often, this is the case with action hooks. When this code fires in WordPress, it looks for every action registered to the `wp_head` hook. It then executes each in order of priority. The page load process continues after every action has completed its execution.

Now look at an action hook that passes extra arguments to any actions attached to it.

```
<?php
do_action( 'save_post', int $post_ID, WP_Post $post, bool
$update );
```

This hook fires when any post in WordPress is saved. It passes a post ID, a post object, and a Boolean value to tell actions whether the post is new or is an update. This can be useful data to have on hand when attaching a custom action to the `save_post` hook.

What Is an Action?

Actions are PHP functions or class methods. What makes it an action is the act of registering it to an action hook. In the previous section, you learned what action hooks are. However, without actions attached to them, the hooks don't do anything on their own.

This is the point where plugins actually do something. When building plugins, you create custom actions that perform a specific task. Then, you attach that action to an action hook. WordPress provides the `add_action()` helper function for handling this.

```
<?php
add_action(
    string $tag,
    callable $function_to_add,
    int $priority = 10,
    int $accepted_args = 1
);
```

This function accepts up to four parameters.

- `$tag`: Name of the action hook tag, which is the first parameter for `do_action()`.
- `$function_to_add`: A PHP callable, such as a function or class method, that executes when the action hook is fired.
- `$priority`: In what order the action should be fired. The default is 10, and negative numbers are allowed. Note that other plugins can set higher or lower priorities than your plugin's actions.
- `$accepted_args`: The number of parameters to pass to your callback function. By default, only the first parameter will be passed to the function if the action has a parameter.

Action hooks can have multiple actions attached to them. Your plugin, other plugins, themes, and even WordPress can add actions to the same hook. For this reason, it's important not to do anything that would interfere with other actions that may execute after yours.

Now you should try creating your first action and attaching it to a hook. `wp_footer` is another common action hook executed on the frontend. It's fired just before the closing `</body>` tag in the HTML output. With the following example code, you'll create a basic plugin that outputs a message that says the site is powered by WordPress:

```
<?php
/**
 * Plugin Name: Footer Message
```

```

 * Plugin URI: http://example.com/
 * Description: Displays a powered by WordPress message in
the footer.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'wp_footer', 'pdev_footer_message', PHP_INT_MAX
);

function pdev_footer_message() {
    esc_html_e( 'This site is powered by WordPress.', 'pdev' );
}

```

Take a look at the `add_action()` call from the example plugin, shown here:

```
add_action( 'wp_footer', 'pdev_footer_message', PHP_INT_MAX
);
```

The function call has three parameters passed into it. `wp_footer` is the action hook tag/name, which tells WordPress what action hook this action is attached to. `pdev_footer_message` is the name of the function that should execute when the action hook is fired. `PHP_INT_MAX` is the priority of the action. It's set to the maximum integer number possible, which tells WordPress that you want your action to execute after everything else (except actions with the same priority, which will execute in order added).

NOTE *Some action hooks in WordPress execute multiple times during a page load. Any callback functions attached to them will fire each time the action hook is called. In some circumstances, you'll need to remove your action or make sure it doesn't run each time.*

Action Hook Functions

When building plugins, you will almost exclusively use `add_action()` or `do_action()` for working with hooks. However, there are times when you'll need to use one of the other functions WordPress provides for working with action hooks.

remove_action()

`remove_action()` is the third most useful function when working with action hooks. It allows you to remove an action that has been previously attached to a hook. At times, you may need to remove actions that WordPress adds. You may also need to remove actions from other plugins, themes, or actions that you've added within your own plugin.

The function returns `true` if the action was successfully removed and `false` if the action isn't registered or otherwise could not be removed.

```
<?php
remove_action( string $tag, callable $function_to_remove,
int $priority = 10 );
```

This function accepts three parameters.

- `$tag`: The name of the hook that you want to remove the action from
- `$function_to_remove`: The name of the callable action to remove from the action hook
- `$priority`: The priority of the action to remove

Each of these parameters must match the parameters passed into the corresponding `do_action()` function call exactly. Otherwise, WordPress will not remove the action from the hook.

Let's take another look at the action from the previous section in this chapter that added a custom message to the footer on the frontend.

```
<?php
add_action( 'wp_footer', 'pdev_footer_message', PHP_INT_MAX
);
```

Now look at the code required to remove that action.

```
<?php
remove_action( 'wp_footer', 'pdev_footer_message',
PHP_INT_MAX );
```

As you can see, the code is almost the same. The only difference is between the function names: `add_action()` and `remove_action()`. If you remember this basic difference, you can remove actions with ease.

You can remove any action added by WordPress, plugin, or theme from within your own plugins. Most often, you'll be removing actions that WordPress adds to change how it works in some way. Many of WordPress' default actions are defined in the `wp-includes/default-filters.php` file. By browsing the code in this file, you'll get a good understanding of how WordPress uses its own action hooks out of the box.

NOTE *Keep in mind that an action cannot be removed before it has been registered. You'll need to make sure your `remove_action()` calls execute after the `add_action()` call of the action you want to remove.*

remove_all_actions()

In rare cases, you may need to remove all actions for a given action hook or all actions with a specific priority on an action hook. More often than not, this may be something like a testing or debugging type of plugin.

WordPress provides the `remove_all_actions()` function to handle this rather than running multiple `remove_action()` calls. This function always returns `true` when it has finished executing.

```
<?php
remove_all_actions( string $tag, int|bool $priority = false
);
```

The `$tag` parameter must be the name of the action hook to remove all actions from. You can set the `$priority` parameter to a specific number, which will remove only those actions that have that priority. Otherwise, the function will remove all actions, regardless of priority.

Imagine that you wanted to stop all scripts, stylesheets, meta tags, or anything else from being output in the `<head>` area on the frontend of the site. You could do so by removing all actions from the `wp_head` hook.

```
<?php
remove_all_actions( 'wp_head' );
```

Let's say you wanted to remove all actions from that hook that had the priority of 1. By default, WordPress adds the following action with that

priority. However, other plugins or the user's theme may also add extra actions on this hook.

```
<?php  
add_action( 'wp_head', '_wp_render_title_tag', 1 );
```

To remove this action as well as all other actions with the same priority, you would use the following code:

```
<?php  
remove_all_actions( 'wp_head', 1 );
```

WARNING *Be extra cautious when using this function. Remember that other plugins and themes may be adding actions that you are unaware of. By removing all actions, you could break functionality that a user expects to work. Unless you have a good reason to do so, it's best to remove specific actions with the remove_action() function instead.*

do_action_ref_array

The `do_action_ref_array()` function is nearly identical to the `do_action()` function. They both create an action hook. The difference is with how arguments are passed. Instead of passing multiple values as the second parameter and beyond, the function accepts an array of arguments. This array is passed along as parameters to any actions attached to the hook.

```
<?php  
do_action_ref_array( string $tag, array $args );
```

Now take a look at one of WordPress' `do_action_ref_array()` calls. The following code snippet shows the core `pre_get_posts` hook, which is fired just before WordPress queries posts from the database on the frontend of the site. It provides a point in the load for plugins to change how posts are loaded.

```
<?php  
do_action_ref_array( 'pre_get_posts', array( &$this ) );
```

You can see from the code that the hook name is `pre_get_posts`. The second parameter is an array with a single item. What this code is doing is

passing along the instance of the `WP_Query` class (`$this`) for the given posts query. This allows plugin authors to directly modify the query.

Imagine that you wanted to build a simple plugin that randomly ordered the posts on the blog home page instead of the default ordering by post date. You would need to register a custom action on this hook and set the ordering.

```
<?php
/**
 * Plugin Name: Random Posts
 * Plugin URI: http://example.com/
 * Description: Randomly orders posts on the home/blog page.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'pre_get_posts', 'pdev_random_posts' );

function pdev_random_posts( $query ) {

    if ( $query->is_main_query() && $query->is_home() ) {
        $query->set( 'orderby', 'rand' );
    }
}
```

has_action

Sometimes you need to check whether a hook has any actions or whether a specific action has been added to a hook prior to executing code. The `has_action()` function is a conditional that returns `true` if an action is found or `false` if no action is found. Like `remove_action()` discussed earlier in this chapter, the check will work only if the action has already been added before your check.

```
<?php
has_action( string $tag, callable|bool $function_to_check =
false );
```

Like most other action-related functions, the first parameter is the hook name that you want to check. The second parameter is optional and defaults to `false`. You can provide the callable if you want to check for a specific function or method attached to the hook.

The return value for `has_action()` can be either a Boolean value or an integer, depending on the scenario. If the `$function_to_check` parameter is set to `false`, the function will return `true` if the hook has any actions or `false` if it has none.

However, if `$function_to_check` is set and the callback has been added to the hook, it will return an integer corresponding to the priority of the action. Otherwise, it will return `false`. Because an action's priority can be `0`, which evaluates to `false`, it's important to use the identical comparison operator (`==`) rather than the equal comparison operator (`==`) in this scenario.

In the next example, you will display a message based on whether any actions have been registered to the `wp_footer` action hook.

```
<?php
if ( has_action( 'wp_footer' ) ) {
    echo '<p>Actions are registered for the footer.</p>';
} else {
    echo '<p>No actions are registered for the footer.
</p>';
}
```

Try out an example where you check for a specific action attached to a hook. The following snippet will check if the `wp_print_footer_scripts` action is running on `wp_footer`. It also assigns the result of `has_action()` to the `$priority` variable. Remember that the result can be `false` or an integer representing the action's priority.

```
<?php
$priority = has_action( 'wp_footer',
'wp_print_footer_scripts' );

if ( false !== $priority ) {
    printf(
        'The wp_print_footer_scripts action has a
priority of %d',
        absint( $priority )
    );
}
```

did_action()

`did_action()` is another conditional function. The name may be confusing at first. You would think that it checks whether a specific action (function)

has executed. However, its purpose is to determine whether an action hook has already run.

```
<?php
did_action( string $tag );
```

The function accepts a single parameter named `$tag`, which should be the tag name for the hook. It returns an integer corresponding to the number of times the action hook has fired (remember that action hooks can fire multiple times). Note that it doesn't return a Boolean value if the hook hasn't yet executed. Instead, it returns `0`, which evaluates to but is not identical to `false`.

The first action hook available to plugins is `plugins_loaded`. Imagine that you wanted to check whether that hook has fired before setting a constant that your plugin is ready.

```
<?php
if ( did_action( 'plugins_loaded' ) ) {
    define( 'PDEV_READY', true );
}
```

current_action

The `current_action()` function returns the name of the action hook that is currently being executed. It is generally useful if you need to use a single action on multiple action hooks but need the context of the hook to alter how something works. It can also be used in any scenario where you might not be sure what the action hook is, such as variable hooks.

For example, assume you have a couple of actions for specific post types on the following core WordPress hook (variable hook names are covered in the “Variable Hooks” section later in this chapter):

```
<?php
do_action( "save_post_{$post->post_type}", $post_ID, $post,
$update );
```

The previous hook name changes depending on context. It could be `save_post_post`, `save_post_page`, or something else entirely. Study the following code that adds a single action to two different action hooks:

```
<?php
add_action( 'save_post_post', 'pdev_check_hook_name' );
add_action( 'save_post_page', 'pdev_check_hook_name' );

function pdev_check_hook_name() {

    $action = current_action();

    if ( 'save_post_post' === $action ) {
        // Do something.
    } elseif ( 'save_post_page' === $action ) {
        // Do something different.
    }
}
```

As you can see, `current_action()` may return different results based on which action hook is currently running. The preceding code uses the return value of the function to set up a conditional statement.

register_activation_hook and register_deactivation_hook

WordPress provides functions for registering actions that will execute when a plugin is activated, deactivated, and even uninstalled. Technically, these functions allow you to create custom hooks and register a callback. These are covered in [Chapter 2](#), “Plugin Framework.”

Commonly Used Action Hooks

Between WordPress, third-party plugins, and themes, a typical page request could run hundreds of action hooks. Even just the hooks provided by core WordPress is more than can be covered in this chapter alone. However, you'll learn some of the more common hooks you'll use when building plugins in this section.

plugins_loaded

The most useful action hook for plugin developers is `plugins_loaded`. It is fired immediately after WordPress has loaded all plugins. It's one of the earliest hooks that plugins can attach an action to and is ideal for running any setup code that your plugin might need. It fires before WordPress loads some of its constants and pluggable functions that plugins are allowed to override too.

Many plugin authors use this hook to set constants or properties on their main plugin class. Let's create a basic plugin that uses this hook and calls a setup class. First, you'll need to create a new folder for your plugin named `plugin-bootstrap`. Then, add a file named `plugin.php` as the primary plugin file with the following code:

```
<?php
/**
 * Plugin Name: Plugin Bootstrap
 * Plugin URI: http://example.com/
 * Description: An example of bootstrapping a plugin.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'plugins_loaded', 'pdev_plugin_bootstrap' );

function pdev_plugin_bootstrap() {

    require_once plugin_dir_path( __FILE__ ) .
'Setup.php';

    $setup = new \PDEV\Setup();

    $setup->boot();
}
```

As you can see, the code adds an action to `plugins_loaded`. The action calls a setup class named `Setup` and calls its `boot()` method. You'll also need to create a `Setup.php` file for the class with the following code:

```
<?php
namespace PDEV;

class Setup {

    public $path;

    public function boot() {

        // Store the plugin folder path.
        $this->path = plugin_dir_path( __FILE__ );

        // Run other setup code here.
    }
}
```

All this code does is store the plugin directory path when the `boot()` method is called. There are many things you can do here, depending on what your plugin needs to do. You don't even need to use a class. There are dozens or hundreds of ways plugins set themselves up using the `plugins_loaded` hook. You'll eventually figure out what best works for you.

init

The `init` hook may be the most-used hook in WordPress. It is fired after most of WordPress is set up. Unlike `plugins_loaded`, themes also have access to this hook. WordPress adds a lot of internal functionality to the hook such as post type and taxonomy registration. This is generally the first hook that will make user data available.

Essentially, if you need to run any code when most of the data and functions are available to you from WordPress, this is the earliest hook you should use.

Imagine that you wanted to add excerpt support for pages in WordPress because the page post type doesn't support this feature by default. You could use the following code to add the feature:

```
<?php
add_action( 'init', 'pdev_page_excerpts' );

function pdev_page_excerpts() {
    add_post_type_support( 'page', [ 'excerpt' ] );
}
```

NOTE *It is arguable that the `init` hook is overused by plugins at times. If there's a more-specific hook available that better suits whatever functionality you're using, be sure to use it instead.*

admin_menu

The `admin_menu` hook is one of the most commonly used WordPress admin hooks. It's fired early in the page load process and is also the proper hook for registering custom admin pages.

The next example adds a basic top-level menu page to the WordPress admin. Typically, you would do something more complex, such as create a settings page (for more information on settings pages, see [Chapter 3](#), “Dashboard and Settings”).

```
<?php
add_action( 'admin_menu', 'pdev_menu_page' );

function pdev_menu_page() {

    add_menu_page(
        'PDEV Page',
        'PDEV Page',
        'manage_options',
        'pdev-page',
        'pdev_menu_page_template'
    );
}

function pdev_menu_page_template() { ?>

    <div class="wrap">
        <h1 class="wp-heading-inline">PDEV Example
Page</h1>

        <p>This is an example admin screen.</p>
    </div>
<?php }
```

save_post

The `save_post` hook executes after a post is saved for the first time or updated. Plugins often use this to store data related to the post, such as post metadata. It's not tied to a specific page, such as the edit post admin screen or the frontend. Posts are saved in a variety of ways at various times, and the `save_post` hook runs every time.

Consider a scenario where you wanted to save some metadata for a post only if it's a new post and not an update. You also don't want to save this metadata for post revisions, which are stored as a separate post. The following code will store the metadata when the `save_post` hook is fired. You'll learn more about metadata in [Chapter 8](#), “Content.”

```

<?php
add_action( 'save_post', 'pdev_save_post', 10, 3 );

function pdev_save_post( $post_id, $post, $update ) {

    if ( $update || wp_is_post_revision( $post_id ) ) {
        return;
    }

    add_post_meta(
        $post_id,
        'pdev_meta_key',
        'This is an example meta value.',
        true
    );
}

```

NOTE *There is also a save_post_{post_type} action hook where {post_type} is the type of post being saved. It is more useful if you need to run code for only a specific post type.*

wp_head

On the frontend of a WordPress site, themes are required to call the `wp_head()` function, which fires the `wp_head` hook. This allows plugins to inject HTML between the opening `<head>` tag and its closing `</head>`.

Typically, plugins use this hook to add SEO features such as meta tags. The following example adds a meta description tag based on the page currently being viewed by the website visitor:

```

<?php
add_action( 'wp_head', 'pdev_meta_description' );

function pdev_meta_description() {

    $description = '';

    // Get site description for front page.
    if ( is_front_page() ) {

        $description = get_bloginfo( 'description',
true );

```

```

// Get post excerpt for singular views.
} elseif ( is_singular() ) {

    $post = get_queried_object();

    if ( $post->post_excerpt ) {
        $description = $post->post_excerpt;
    }
}

if ( $description ) {
    printf(
        '<meta name="description"
content="%s"/>',
        esc_attr( wp_strip_all_tags(
$description ) )
    );
}
}

```

There are dozens of SEO plugins out there, but don't let that discourage you from using the previous code to kick-start your own competing product and gain a foothold on the market.

WARNING *Often, plugins incorrectly use the wp_head action hook to load JavaScript in the header. WordPress provides the wp_enqueue_script() function and more appropriate hooks for loading scripts and styles. You'll learn about this in [Chapter 6](#), “JavaScript.”*

FILTERS

Filter hooks are the second type of hook in WordPress. They are similar to action hooks in some ways. However, they are much different in their most essential feature, which is to allow you to manipulate the output of code. Filter hooks work by always passing data through the hook, and any filters on the hook can modify the data in any way.

To understand how filter hooks work, you must first learn how the `apply_filters()` WordPress function works. It's the foundation of the filter hook system.

```
<?php
apply_filters( string $tag, mixed $value );
```

The function accepts two or more parameters.

- \$tag: The unique name for the filter hook.
- \$value: The value that gets passed to any filters added to the hook for manipulation.
- \$args: Like the `do_action()` function for action hooks, the function accepts any number of additional arguments to pass to filters attached to the hook.

The function always returns the `$value` variable after it has been filtered. It is also imperative that any filters added to the hook also return a value.

The following is an example of a filter hook in core, which should give you a feel for what they look like.

```
<?php
$template = apply_filters( 'template_include', $template );
```

In this example, `template_include` is the name of the filter hook.

`$template` is a string representing a file path for the template (PHP file) to load on the frontend of the site to display the current page. A plugin could filter this file path to let WordPress know it should load the template from a different location.

What Is a Filter?

Like actions covered earlier in this chapter, filters are nothing more than a function or other PHP callable. What makes the function a filter is the act of registering it for a filter hook. If a filter hook doesn't have any filters on it, nothing happens. The data is returned unchanged. Filter hooks exist so that plugin developers can create custom functions that filter some type of data in some way. This can be a simple text string, an array, or any other valid data type in PHP.

When the `apply_filters()` function is called, any filters attached to it are executed in the order of their priority. You use the `add_filter()` function to register a filter.

```
<?php
add_filter(
    string $tag,
    callable $function_to_add,
    int $priority = 10,
    int $accepted_args = 1
);
```

This function accepts up to four parameters.

- `$tag`: The name of the filter hook tag, which is the first parameter for `apply_filters()`.
- `$function_to_add`: A PHP callable, such as a function or class method, that executes when the filter hook is applied.
- `$priority`: The order in which the filter should be applied to the hook. The default is 10, and negative numbers are allowed.
- `$accepted_args`: The number of parameters to pass to your callback function. By default, the `$value` parameter from `apply_filters()` will always be passed as the first parameter.

You can add any number of filters to the same filter hook. WordPress itself or third-party plugins may also have filters applied to the same hook. It's important to understand this because each filter can manipulate the data in any way it wants. When adding a filter, you must always return the same type of data that is expected for a hook. Otherwise, you risk breaking other plugins or even WordPress, which creates a poor experience for plugin users.

Now look at an example filter hook from WordPress. It is the `body_class` hook that applies to the HTML classes in the `<body>` element on the frontend of the site.

```
<?php
$classes = apply_filters( 'body_class', $classes, $class );
```

There are three things you should note about this hook.

- `body_class`: The name of the hook.

- `$classes`: An array of HTML classes. Filters must return a manipulated version of this or an empty array.
- `$class`: A user-submitted class passed as a second parameter if needed for filters.

Now you're going to write a basic function that adds class to the `$classes` array and returns it.

```
<?php
add_filter( 'body_class', 'pdev_body_class' );

function pdev_body_class( $classes ) {

    $classes[] = 'pdev-example';

    return $classes;
}
```

The preceding filter is simple. It adds the `pdev-example` class to the array. WordPress will later format that array to a string and output it as a class attribute to the `<body>` element. You could run any code within the filter to determine whether you want to add a class. For example, you may want to add a class only on certain pages on the frontend related to your plugin.

Filter Hook Functions

You've learned how `apply_filters()` and `add_filter()` work, which are the basics of using filters. For the most part, you will almost always use these two functions. However, there are cases where you will need to utilize the other filter-related functions provided by WordPress.

remove_filter

One of the more important functions related to filters is the `remove_filter()` function. It allows plugins to remove filters that are registered to a filter hook but only if they have been registered earlier in the page load than the `remove_filter()` function call.

```
<?php
remove_filter( string $tag, callable $function_to_remove,
int $priority = 10 );
```

The function accepts three parameters.

- `$tag`: The name of the filter hook to remove the filter from
- `$function_to_remove`: The callback function or method to remove from the filter hook
- `$priority`: The exact priority of the filter added to the hook

The function will return `true` when the filter is successfully removed and `false` otherwise. Note that the `$priority` parameter must match the `$priority` of the original `add_filter()` call for the filter. Otherwise, it will not be removed.

Now take a look at one of WordPress' default filters, which is defined in `wp-includes/default-filters.php`. This particular filter converts smiley characters such as `:)`, `:)`, and `;` into images for post content on the frontend.

```
<?php
add_filter( 'the_content', 'convert_smilies', 20 );
```

Perhaps you want to disable these smiley images because you don't think they look professional or for some other reason. By simply looking at the `add_filter()` call, you have all the information you need to remove it: the hook name, the function name, and the priority. Now try the following code to disable smiley-to-image conversion:

```
<?php
remove_filter( 'the_content', 'convert_smilies', 20 );
```

As you can see, the `add_filter()` and `remove_filter()` calls are nearly identical. You shouldn't have any trouble removing filters after trying it a couple of times.

remove_all_filters

At times, you may need to remove all filters from a filter hook or just all filters with a specific priority. The `remove_all_filters()` function allows you to do this with a single line of code. Similar to `remove_filter()`, it can remove only the filters that have already been registered at the time the code runs.

```
<?php
remove_all_filters( string $tag, int|bool $priority = false
);
```

The function accepts a `$tag` parameter, which must be the tag name for the hook to remove filters from. The second parameter, `$priority`, is optional. If set to false (the default), it will remove all filters from the hook. If set to a specific integer, it will remove only those filters that have a matching priority.

After the function has completed running, it will always return `true`. Note that it doesn't return `false` if no filters were removed.

Now take a look at this list of the filters that core WordPress runs over its `the_content` filter hook:

```
<?php
add_filter( 'the_content', 'do_blocks', 9 );
add_filter( 'the_content', 'wptexturize' );
add_filter( 'the_content', 'convert_smilies', 20 );
add_filter( 'the_content', 'wpautop' );
add_filter( 'the_content', 'shortcode_unautop' );
add_filter( 'the_content', 'prepend_attachment' );
add_filter( 'the_content',
'wp_make_content_images_responsive' );
```

Imagine that you wanted to remove all those filters and any others registered for the filter hook. You'd use the following code:

```
<?php
remove_all_filters( 'the_content' );
```

Now suppose you wanted to remove only those filters with a priority of 9 (the first filter in the earlier code block has this priority). You would need to set the second parameter, as shown here:

```
<?php
remove_all_filters( 'the_content', 9 );
```

WARNING *Remember that you can never be 100 percent sure what filters are registered for a specific hook unless you have full control over the environment. You should rarely remove all filters from a hook or you risk breaking third-party plugins.*

apply_filters_ref_array

The `apply_filters_ref_array()` function is similar to the primary `apply_filters()` function. It creates a filter hook. However, the major difference is that it accepts an array of arguments to pass to filters.

```
<?php
apply_filters_ref_array( string $tag, array $args );
```

The `$tag` parameter is the name of the hook. The `$args` parameter is the array of arguments, which will be split up in order as parameters to pass to any filter callback functions.

Take a look at the following example filter hook from core WordPress' `WP_Query` class, which returns an array of posts that should be displayed for a given page.

```
<?php
$this->posts = apply_filters_ref_array(
    'the_posts',
    array( $this->posts, &$this )
);
```

In reality, this would not be any different from the equivalent code using `apply_filters()`.

```
<?php
$this->posts = apply_filters( 'the_posts', $this->posts,
&$this );
```

Adding custom filters to this type of filter hook is no different. Imagine that you wanted to remove the first post from the posts list. Perhaps you're showing it in a different way elsewhere on the page. You'd use the following code to remove the first post from the array:

```
<?php
add_filter( 'the_posts', 'pdev_the_posts' );

function pdev_the_posts( $posts ) {

    if ( isset( $posts[0] ) ) {
        unset( $posts[0] );
    }

    return $posts;
}
```

has_filter

`has_filter()` is a conditional function that allows plugin authors to check whether any filters have been registered or whether a specific filter has been registered for a filter hook. Similar to `remove_filter()`, the function will recognize a filter as registered only if the call to `has_filter()` comes after `add_filter()` in the page load process.

```
<?php
has_filter( string $tag, callable|bool $function_to_check =
false );
```

The first parameter of `$tag` should be the name of the hook to check against. The second parameter of `$function_to_check` is optional. If omitted, the function will check whether any filters are registered for the hook. If a specific function is passed, the function will check whether that function is registered as a filter.

The function will return different values based on whether `$function_to_check` is set to `false` (the default) or a function. If checking against a specific function, the `$priority` parameter passed into `add_filter()` will be returned if it is registered. Note that priorities can have a value of `0`, so it's important to perform identical evaluation (`==`) instead of equals evaluation (`==`). If the filter is not registered to the hook, the function will return `false`.

If no `$function_to_check` parameter is set, the function will return either `true` or `false`, depending on whether any filters are registered for the hook.

Suppose you wanted to check whether there were any filters on the post content and display a message if any were found. You'd use the following

code:

```
<?php
if ( has_filter( 'the_content' ) ) {
    echo 'There are filters on the post content.';
}
```

You could also get the priority of a specific filter on the post content, such as determining when the editor blocks are parsed, using the next code snippet.

```
<?php
$priority = has_filter( 'the_content', 'do_blocks' );

// Returns:
// 9
```

current_filter

The `current_filter()` function returns the name of the filter hook that is currently being executed. This function is particularly useful if you use a single filter function on multiple filter hooks. There are cases where you might want the function to execute differently depending on the current filter hook.

Imagine that you had a client who wanted to replace specific “bad” words from post titles and post content. However, the bad words are different depending on context. You could write two different filter functions, or you could combine them into one and use `current_filter()` to determine which filter hook (the post title or post content hook) is running.

With the following code, you could write a plugin that replaces any of the bad words with *** so that they don't appear on the site:

```
<?php
/**
 * Plugin Name: Remove Bad Words
 * Plugin URI: http://example.com/
 * Description: Removes bad words from the post title and
content.
 * Author:      WROX
 * Author URI:  http://wrox.com
 */
```

```

add_filter( 'the_title', 'pdev_remove_bad_words' );
add_filter( 'the_content', 'pdev_remove_bad_words' );

function pdev_remove_bad_words( $text ) {

    $words = [];

    if ( 'the_title' === current_filter() ) {
        $words = [
            'bad_word_a',
            'bad_word_b'
        ];
    } elseif ( 'the_content' === current_filter() ) {
        $words = [
            'bad_word_c',
            'bad_word_d'
        ];
    }

    if ( $words ) {
        $text = str_replace( $words, '***', $text );
    }

    return $text;
}

```

Quick Return Functions

There are times when you may need to return values like an empty array, empty string, or similar simple types of simple data with filters. WordPress provides several quick return functions so that you don't have to create a function to handle it.

For example, the following is an example of returning an empty array with a filter to remove all user contact methods:

```

<?php
add_filter( 'user_contactmethods', 'pdev_return_empty_array' );

function pdev_return_empty_array() {
    return [];
}

```

There's no need to write a custom function in that scenario. Instead, you could use the core WordPress `__return_empty_array()` function and

handle the filter with a single line of code.

```
<?php  
add_filter( 'user_contactmethods', '__return_empty_array' );
```

WordPress provides several quick return functions for various scenarios.

- `__return_true()`: Returns a Boolean true value
- `__return_false()`: Returns a Boolean false value
- `__return_zero()`: Returns the integer 0
- `__return_empty_array()`: Returns an empty array
- `__return_null()`: Returns a null value
- `__return_empty_string()`: Returns an empty string

As you've probably noticed, each of these function names begin with a double underscore. This is a quick way to recognize quick return functions in WordPress. If there's not a common function available for returning a specific value, you can also create your own for use in your plugins. Core WordPress simply covers some of the most common use cases.

Commonly Used Filter Hooks

WordPress has hundreds of filter hooks that may be fired on any given page. There's no way this book could cover each of them. New filter hooks are added over time too. This section covers a few of the more common filter hooks to provide a look at what is possible with custom filters.

the_content

There's one hook that is used more than any other by plugin authors, which is `the_content`. It's one of the most important hooks on most frontend pages of the site because it allows a plugin to filter the output of the post content. Plugins typically do one of two things on this hook: add extra formatting to the existing content or append extra HTML to the end of the content.

You will now build a plugin that adds a subscription form to the end of single post content on the frontend.

```

<?php
/**
 * Plugin Name: Content Subscription Form
 * Plugin URI: http://example.com/
 * Description: Displays a subscription form at the end of
 * the post content.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_filter( 'the_content', 'pdev_content_subscription_form',
PHP_INT_MAX );

function pdev_content_subscription_form( $content ) {

    if ( is_singular( 'post' ) && in_the_loop() ) {

        $content .= '<div class="pdev-subscription">
                    <p>Thank you for reading.
                    Please subscribe to my email list for
        updates.</p>
                    <form method="post">
                        <p>
                            <label>
                                Email:
                                <input type="email"
value="" />
                            </label>
                        </p>
                        <p>
                            <input type="submit"
value="Submit" />
                        </p>
                    </form>
                </div>';
    }

    return $content;
}

```

Of course, you'd need to connect that form to an email or feed system for it to actually work. The preceding code is simply to show how to append such a form to the end of the content.

template_include

The `template_include` hook is a hook where plugins intersect with the theme system by overwriting or changing the template included (templates are just PHP files). At times, it's not a perfect system because themes are in ultimate control of what templates exist and conflicts arise. However, plugins filtering the included template can minimize any damage.

This particular filter hook is a catchall hook for many other, more-specific filter hooks related to templates. It is fired after the theme template file has been chosen for the current page view.

Suppose you had a custom post type named `movie` in which you wanted to use a template defined by the plugin if the template doesn't exist in the theme (custom post types are covered in [Chapter 8](#), “Content”). You want to check whether the theme has a `pdev-movie-archive.php` template for the post type archive and a `pdev-single-movie.php` for the single movie post. If not found, you want to use the templates stored in your plugin's custom `/templates` folder.

```
<?php
add_filter( 'template_include', 'pdev_template_include' );

function pdev_template_include( $template ) {
    if ( is_post_type_archive( 'movie' ) ) {
        $template = locate_template( 'pdev-movie-
archive.php' );
        if ( ! $locate ) {
            $template = require_once
plugin_dir_path( __FILE__ )
                    . 'templates(pdev-movie-
archive.php';
    }

    } elseif ( is_singular( 'movie' ) ) {
        $template = locate_template( 'pdev-single-
movie.php' );
        if ( ! $locate ) {
            $template = require_once
plugin_dir_path( __FILE__ )
                    . 'templates(pdev-single-
```

```
movie.php';
        }
    }

    return $template;
}
```

You could also split these filters into two separate functions and register them for the `archive_template` and `single_template` filter hooks. However, those hooks fire earlier in the page load process.

USING HOOKS FROM WITHIN A CLASS

Throughout most of this chapter, you've seen code examples of using action and filters with PHP functions. This makes it simpler to show how actions and filters work. However, in most professional plugin projects, you'll want to use PHP classes to build your plugins. When adding a class method as an action or filter, the format of the `add_action()` and `add_filter()` calls will be slightly different.

You will primarily be working with object instances in object-oriented programming. Therefore, you need a reference to the object (`$this` from within the class). When working with objects, the format of the function calls will look like the following:

```
<?php
add_action( $tag, [ $object, $method ] );
add_filter( $tag, [ $object, $method ] );
```

The second parameter becomes an array with the object reference and the method name of the class.

Now you will rebuild the action from earlier in this chapter that created random posts on the blog page. Instead of using a basic function, you will do it from within a class.

```
<?php
namespace PDEV;

class RandomPosts {

    public function boot() {
        add_action( 'pre_get_posts', [ $this,
```

```

'randomize' ] );
}

public function randomize( $query ) {

    if ( $query->is_main_query() && $query-
>is_home() ) {
        $query->set( 'orderby', 'rand' );
    }
}

( new RandomPosts() )->boot();

```

NOTE *Many plugin authors add their add_action() or add_filter() function call within a class' __construct() method. Generally, this is poor practice because class constructors are meant for setting the initial state of the object. Instead, you should create a boot(), init(), or similar method to “bootstrap” your actions/filters.*

At times, you may also need to run an action or filter from within a static class. Because you don't have an object reference, you need to reference the class name instead. Take a look at the same random posts code from within a static class:

```

<?php
namespace PDEV;

class RandomPosts {

    public static function randomize( $query ) {

        if ( $query->is_main_query() && $query-
>is_home() ) {
            $query->set( 'orderby', 'rand' );
        }
    }

    add_action( 'pre_get_posts', [ 'PDEV\RandomPosts',
'randomize' ] );
}

```

Take note that the code references the fully qualified class name. WordPress won't recognize your namespace in `add_action()` and `add_filter()` calls, so it's important to use the full class name. You can also use `__NAMESPACE__` to get the current namespace.

USING HOOKS WITH ANONYMOUS FUNCTIONS

There are times when you may want to use an anonymous function (closure) with an action or filter. Typically, these are quick functions where it doesn't make sense to build out a full, named function. They can also be useful where you want to run an action/filter that you don't want other plugins to remove.

For example, take a look at the following code that runs on the `plugins_loaded` hook for setting up a plugin. Small plugins may not need a large setup class and simply want to jump-start some quick setup code.

```
<?php
add_action( 'plugins_loaded', function() {
    // Run plugin setup.
} );
```

As you can see, the second parameter registering an anonymous action or filter is simply the anonymous function itself. There's no reference to a function name, class name, or object.

WARNING *Anonymous functions cannot be unhooked by other plugins. Part of building plugins means playing nicely with others and making sure your code can be manipulated through the Plugin API. For this reason, it's usually not a good idea to add actions and filters that other plugins cannot remove.*

CREATING CUSTOM HOOKS

One of the great things about hooks in WordPress is that plugin developers are not limited to the hooks that WordPress creates. Plugins can contain

custom hooks using the standard functions covered earlier in this chapter.

- `do_action()`
- `do_action_ref_array()`
- `apply_filters()`
- `apply_filters_ref_array()`

You can use the first two functions for creating custom action hooks and the last two for creating custom filter hooks just like core WordPress.

Benefits of Creating Custom Hooks

Custom hooks make your plugin more flexible by allowing it to be extended by other plugin and theme developers. It also gives you the ability to execute code or filter data within the plugin itself.

One of the biggest advantages of using the hook system is to keep users or developers from editing your plugin's code, which makes it incompatible with future updates. With hooks, others can create custom actions and filters to modify how your plugin works without losing their modifications. It's always important to think about what data could be modified via a filter or what event might make for a good action filter when writing your plugin's code.

Custom Action Hook Example

In the following example, you create a plugin setup/load function, which is hooked to `plugins_loaded`. This code adds an action hook before and after your plugin's setup runs. Doing this provides an easy way for other plugins to run code at those specific points.

```
<?php
add_action( 'plugins_loaded', 'pdev_load' );

function pdev_load() {

    do_action( 'pdev_load_before' );
    // Run setup code.
```

```
        do_action( 'pdev_load_after' );
    }
```

Imagine you had a large plugin with those action hooks where other plugin developers created “add-on” plugins that extended your plugin. The `pdev_load_after` action hook would be a good point for the add-ons to set themselves up. It also makes it easy for them to run code only if your plugin is active. The following code snippet shows how such a plugin might set itself up on that hook:

```
<?php
add_action( 'pdev_load_after', 'pdev_addon_load' );

function pdev_addon_load() {
    // Add-on plugin sets itself up.
}
```

Custom Filter Hook Example

Suppose you have a form in your plugin that saves some text that will eventually get output on the frontend. By default, this text has no formatting. You might want to allow others to filter the final output of the text.

```
<?php
function pdev_example_text() {

    $text = apply_filters(
        'pdev_example_text',
        'This is some example text the user has
    saved.',
    );

    echo $text;
}
```

Another plugin could handle a feature such as adding automatic paragraph tags using the WordPress `wpautop()` function by adding it as a filter on the `pdev_example_text` hook.

```
<?php
add_filter( 'pdev_example_text', 'wpautop' );
```

A developer could also write a custom filter function just like with any hook and do any sort of formatting. There are no real limits to what can be done.

FINDING HOOKS

It would be nearly impossible to list every hook available in WordPress in this book. You learned about some of the most common action and filter hooks earlier in this chapter, but those sections cover a small sampling of what hooks WordPress offers to plugin developers.

New hooks are almost always added with new versions of WordPress. As a developer, you'll want to keep track of changes in the core code with each update, which will help you stay on top of new hooks as they become available.

Searching for Hooks in the Core Code

The best thing any developer at any level can do is familiarizing themselves with the code of the software they're building on. Books such as *Professional WordPress Plugin Development* can help you learn the “how” of building plugins, but nothing beats simply reading the code for learning how to program for that system.

The easiest way to start finding hooks is to open WordPress in your preferred code editor and use its search or find function. Using this feature of your editor, you merely need to search for one of the following strings.

- `do_action`
- `do_action_ref_array`
- `apply_filters`
- `apply_filters_ref_array`

Once you get a hit with your editor's search, you might feel overwhelmed by the number of results that turn up. Take your time and read the code. You have the knowledge of how hooks work. At this point, it is a matter of figuring out what the specific hook does.

Variable Hooks

When searching for hooks in the core WordPress code, you will eventually come across what is called *variable hooks*. Throughout this chapter, you have only seen hooks with a static string of text for their name, which is the most common case. However, some hooks use PHP variables in their names, which can change the hook name depending on the variable's current value.

The `save_post_{$post->post_type}` hook is a good example of such a hook. The `$post->post_type` variable changes depending on what type of post is currently being saved.

```
<?php
do_action( "save_post_{$post->post_type}", $post->ID, $post,
true );
```

Because the variable changes depending on the post type, the real hook name could be `save_post_post` (“post” post type), `save_post_page` (“page” post type), `save_post_movie` (a custom “movie” post type), or any other post type name.

WordPress has several variable hooks in the core code. These hook names are useful because they provide context to plugin developers, which allows them to execute only under specific circumstances. Keep this in mind when searching for the perfect hook to use for your plugin.

Hook Reference Lists

Searching for hooks in the core code is a great way to learn, but sometimes you may find it easier to access the publicly available references in the WordPress developer documentation. These references can save you time and may also have useful notes.

- developer.wordpress.org/reference/hooks
- codex.wordpress.org/Plugin_API/Action_Reference
- codex.wordpress.org/Plugin_API/Filter_Reference

[Chapter 16](#), “The Developer Toolbox,” has additional reference materials for plugin authors to use when building plugins.

SUMMARY

The Plugin API is the most important aspect of building WordPress plugins. With nearly every plugin you build, your plugin will hook its functions into one or more WordPress action or filter hooks. You'll make extensive use of everything you have learned in this chapter in your journey of building plugins. In many ways, the hooks system is what sets WordPress apart from many other platforms. It provided a leg up over the competition early on because it was built to be extended, and hooks are nothing more than a way for plugins to extend WordPress.

Armed with a full understanding of how hooks work, you can now start building real plugins.

6

JavaScript

WHAT'S IN THIS CHAPTER?

- Introduction to JavaScript
- Registering scripts
- Enqueueing scripts
- Limiting scope
- Localizing scripts
- Inline scripts
- Overview of bundled scripts
- Polyfills
- Your custom scripts
- Old reliable: jQuery
- Middle reliable: Backbone and Underscore
- New reliable: React

In the first edition of this book, we wrote that JavaScript is “principally a language used to code plain-text script executed on the client side, that is, the browser.” Since then, JavaScript has exploded into one of the world’s most popular programming languages, and that statement is no longer completely accurate.

JavaScript can be used to power a web server using Node, and the web is now full of rich and powerful tooling that is pushing the boundaries of what is possible in-browser. Even the bulk of the latest and greatest block-based content editor is coded using nearly nothing but JavaScript.

This chapter covers best practices for getting your JavaScript files registered and enqueued inside WordPress. It will educate you on what

JavaScript comes bundled with WordPress, including some oldies but goodies. It will help prepare you for the next chapter, which will put everything you've learned across multiple chapters to work using Gutenberg, the block-based editing experience included in the latest versions of WordPress.

When it comes to JavaScript in WordPress, remember this:

Always register. Conditionally enqueue.

REGISTERING SCRIPTS

WordPress expects all JavaScript files to be registered. Registering prevents multiple plugins from including duplicates of the same JavaScript libraries multiple times and also just generally helps you keep track of all the JavaScript you are packaging inside your WordPress plugins (which can sometimes be a lot!).

Registering is simply announcing that some unique JavaScript file exists so that it can be easily included in the correct order later. Many things in WordPress are registered before they are used. Things like Post Types and Taxonomies are good examples of things we've already touched on in this book, and JavaScript is no different.

At the time of this writing, the official WordPress documentation considers registering your JavaScript files as optional, but we strongly recommend you consider it a requirement for everything you do. Even if you decide to skip this step and jump ahead to enqueueing, if you enqueue something that isn't registered, it does the registration just-in-time for you anyway. Do it yourself, and you'll be glad you did.

You can register your custom JavaScript files anytime in your loading or bootstrapping process. We recommend doing this relatively early, like on the `plugins_loaded` or `init` action hook, because if they are registered too late, it makes it difficult for other plugins to use your JavaScript reliably.

Register your JavaScript via the `wp_register_script()` function, like so:

```
/**  
 * Register a script  
 */
```

```

function pdev_register_scripts() {
    wp_register_script(
        'pdev-your-script-id', // Unique name
        plugin_dir_url( __FILE__ ) . 'your.js', // URL to the file
        array(), // Dependencies
        '1.0.0', // Version
        true // Output in Footer
    );
}
add_action( 'init', 'pdev_register_scripts' );

```

You can also similarly deregister a script if you need to:

```

/**
 * Deregister a script
 */
function pdev_deregister_scripts() {
    wp_deregister_script( 'pdev-your-script-id' );
}
add_action( 'init', 'pdev_deregister_scripts' );

```

We recommend registering each of your unique JavaScript files (and to avoid using the JavaScript `@import` approach) because it will always be better for WordPress to be aware of your JavaScript intentions, plus it helps you better organize all of the JavaScript you will likely be writing and working with in your professional plugins going forward.

It is still possible to write inline JavaScript, but it is highly recommended you avoid going this route in your professional plugins and stick to registering them. If you're considering just dumping some `<script>` tags into your post content or `<head>`, please reconsider refactoring your code so that your JavaScript fits into individual files that can be registered; you will be glad you did!

ENQUEUEING SCRIPTS

Now that you've registered your JavaScript files, it's time to enqueue them. Enqueueing is the way that WordPress manages what will end up being a

somewhat complex tree of relationships between different scripts, the files they depend on, and the files that depend on them. Under the hood, it uses the `WP_Dependencies` API, which is exactly what is used when registering and enqueueing CSS.

This is also how you tell WordPress that your intention is to interact with (or include) what JavaScript on which available pages. Thanks to having registered all of your JavaScript earlier, the dependency tree will be completely managed for you!

You will typically want to enqueue your JavaScript on one of two different WordPress action hooks: `admin_enqueue_scripts` or `wp_enqueue_scripts`. The former is for admin area JavaScript, and the latter is for theme-side JavaScript.

There are other atypical reasons and ways to enqueue your JavaScript, but we recommend avoiding them as much as you can. The added code complexity is difficult to maintain over time and will cause confusion and misunderstandings when it comes time to share your code with colleagues and with the rest of the world.

You will enqueue your registered JavaScript like this:

```
/**
 * Enqueue a script
 */
function pdev_enqueue_scripts() {
    wp_enqueue_script( 'pdev-your-script-id' );
}
add_action( 'wp_enqueue_scripts', 'pdev_enqueue_scripts' );
```

You may also similarly dequeue a previously enqueued script.

```
/**
 * Dequeue a script
 */
function pdev_dequeue_scripts() {
    wp_dequeue_script( 'pdev-your-script-id' );
}
add_action( 'wp_enqueue_scripts', 'pdev_dequeue_scripts' );
```

LIMITING SCOPE

Now that you know how to register and enqueue your JavaScript, the next thing you'll want to do is tune it so that you are enqueueing only on the WordPress pages where it is necessary for that JavaScript to be.

This is always going to be different based on the needs of your application, but by using your best judgment and a bit of trial and error, you will find it is possible to narrow down the scope based on just about any criteria you can imagine.

WordPress includes many helper functions for identifying what kind of page is being loaded. Functions like `is_singular()`, `is_archive()`, and more can be used to tell WordPress whether your custom JavaScript should be included.

In this example, you are looking for single pages with gallery shortcodes only:

```
/**  
 * Enqueue only on single pages using Gallery shortcodes  
 */  
function pdev_enqueue_gallery_scripts() {  
    if ( is_singular() && has_shortcode( get_the_content(), 'gallery' ) {  
        wp_enqueue_script( 'pdev-your-script-id' );  
    }  
}  
add_action( 'wp_enqueue_scripts',  
    'pdev_enqueue_gallery_scripts' );
```

LOCALIZING SCRIPTS

When you want to take something out of PHP and make it available within your JavaScript, you need to localize it. This API gets its name from the idea that all languages (not just coding ones) are restricted to their particular fixed area, but sometimes it's necessary to translate something from one language to another.

In the normal world, the act of translating words between languages is called *localization*, so why would that be any different in the coding world? Just because you know something in PHP doesn't mean you shouldn't know it in your JavaScript too, right? Right!

Localizing allows you to bridge the gap between what variable content you can ask WordPress for in PHP and translate it into your custom JavaScript for later reuse. In this section, you will take a PHP array of keys and values and turn it into a JavaScript object of keys and values instead. Magic!

In this example, we are localizing some strings and the time into our JavaScript:

```
/**  
 * Enqueue only on single pages using Gallery shortcodes  
 */  
function pdev_localize_scripts() {  
    wp_localize_script(  
        'pdev-your-script-id', // Script  
        handle from previous  
        'pdevScript', // Name of  
        JavaScript object  
        array( // Array of  
            properties for JS object  
            'greeting' => __( 'Hello' ),  
            'repeat' => __( 'Hello, again' ),  
            'time' => time()  
        )  
    );  
}  
add_action( 'wp_enqueue_scripts', 'pdev_localize_scripts' );
```

You can include anything you need to in your array, but you'll need to remember that localization applies only to the registered and enqueueed JavaScript referenced in the script handle and only by using the name of the JavaScript object as you've named it. If you have more than one file you need to localize, simply make multiple calls to `wp_localize_script()`, one for each file.

INLINE SCRIPTS

If you must include inline scripts to accomplish your goals, you should use the `wp_add_inline_script()` function to do so. This allows you to output JavaScript directly before or after a registered JavaScript file is rendered to the page, allowing you to sneak ahead of or behind it right away, rather than needing to worry about the dependency tree of multiple JavaScript files.

In many ways, you can probably get by without using this approach, but it is useful for the times when you do need to do this. A good example of this in WordPress is the `wp_localize_jquery_ui_datepicker()` function. Its job is to make sure that anytime the jQuery UI Datepicker is initialized, the date and time abbreviations for the currently selected language are set in JavaScript.

```
/***
 * Localizes the jQuery UI datepicker.
 *
 * @since 4.6.0
 *
 * @link https://api.jqueryui.com/datepicker/#options
 *
 * @global WP_Locale $wp_locale WordPress date and time
 * locale object.
 */
function wp_localize_jquery_ui_datepicker() {
    global $wp_locale;

    if ( ! wp_script_is( 'jquery-ui-datepicker',
        'enqueued' ) ) {
        return;
    }

    // Convert the PHP date format into jQuery UI's
    // format.
    $datepicker_date_format = str_replace(
        array(
            'd',
            'j',
            'l',
            'z', // Day.
            'F',
            'M',
            'n',
            'm', // Month.
            'Y',
            'y', // Year.
        ),
        array(
            'dd',
            'd',
            'DD',
            'o',
            'MM',
        )
    );
}
```

```

        'M',
        'm',
        'mm',
        'yy',
        'y',
    ),
    get_option( 'date_format' )
);

$datepicker_defaults = wp_json_encode(
    array(
        'closeText'      => __( 'Close' ),
        'currentText'    => __( 'Today' ),
        'monthNames'     => array_values(
$wp_locale->month ),
        'monthNamesShort' => array_values(
$wp_locale->month_abbrev ),
        'nextText'        => __( 'Next' ),
        'prevText'        => __( 'Previous' ),
        'dayNames'        => array_values(
$wp_locale->weekday ),
        'dayNamesShort'   => array_values(
$wp_locale->weekday_abbrev ),
        'dayNamesMin'     => array_values(
$wp_locale->weekday_initial ),
        'dateFormat'      =>
$datepicker_date_format,
        'firstDay'        => absint( get_option(
'start_of_week' ) ),
        'isRTL'           => $wp_locale->is_rtl(),
    )
);

wp_add_inline_script( 'jquery-ui-datepicker',
    "jQuery(document).ready(function(jQuery)
{jQuery.datepicker.
    setDefaults({$datepicker_defaults});});" );
}

```

As you can see, `wp_add_inline_script()` is particularly useful here in making sure that no matter when or why you are using the jQuery UI Datepicker library, it will always be invoked with the correct strings and settings for the chosen language.

At the time of this writing, you should consider using this in your own plugins relatively sparingly. The majority of the time, you will have

complete control over your own code, where you should be able to register, enqueue, and localize, without ever having the need to use inline JavaScript. But as the web becomes increasingly dependent on JavaScript, inline scripts are becoming increasingly popular.

Here's a quick `console.log()` to show you how to do it:

```
/**  
 * Inline right after your other script has been output to  
 * the page  
 */  
function pdev_inline_scripts() {  
    wp_add_inline_script (  
        'pdev-your-script-id', // Script handle from  
        previous  
        'console.log("hello")' // Your raw JavaScript  
    );  
}  
add_action( 'wp_enqueue_scripts', 'pdev_inline_scripts' );
```

OVERVIEW OF BUNDLED SCRIPTS

WordPress comes bundled with many extremely useful JavaScript libraries and tools. These tools are there to be used, and you should at all costs avoid inventing your own versions of these technologies (unless you have a specific reason to do so).

The reason so much JavaScript is bundled with WordPress is because WordPress itself uses a lot of JavaScript for many everyday functions that you may not have ultimately needed to consider. Admin area pages like Menus, the Customizer, the Media Library, and even the Dashboard all rely on JavaScript libraries to make them work.

As time goes on, many pieces of functionality are being rewritten using React, but it will be many years before jQuery is completely unplugged from WordPress Admin, likely requiring a complete ground-up rewrite of the entire Dashboard.

jQuery UI and Other Scripts

The bulk of the bundled scripts belongs to a library called jQuery and its partner project jQuery UI. Together, these scripts account for nearly 50

different unique JavaScript files that all exist to help make many complex tasks a breeze.

The list of bundled JavaScript changes rarely, but it does regularly enough that including a comprehensive list of them here would likely be out-of-date relatively quickly, so rather than list them all, we'll show you how to enqueue them quickly.

Let's say you want to include the jQuery UI library Sortable, because you have a need to make certain aspects of your plugin UI be able to be rearranged by the user. Thanks to the fact that all of these bundled libraries are already registered inside WordPress, you would simply need to enqueue it like so:

```
/**  
 * Enqueue only on single pages using Gallery shortcodes  
 */  
function pdev_enqueueSortable_scripts() {  
    wp_enqueue_script( 'jquery-ui-sortable' );  
}  
add_action( 'wp_enqueue_scripts',  
    'pdev_enqueueSortable_scripts' );
```

You may have noticed that we never enqueue jQuery itself. That's because WordPress already registered it as a dependency and knows that if you want jQuery UI Sortable, it needs to include jQuery ahead of it in the first place or your Sortables will not work.

In addition to jQuery, there are multiple color pickers, image croppers, Backbone, Underscore, zxvcvbn, Masonry, CodeMirror, Twemoji, and many other useful tools to help you get a jump-start on your plugin development.

The WP Global

The `wp` JavaScript global is a monolithic object that is used to encapsulate a ton of extremely useful utilities. Behind the scenes, WordPress has been using this global for years now as the place where it attaches most of its JavaScript-related needs.

In recent years, this global has become the home to emoji parsing, the Heartbeat API, JavaScript Hooks, svgPainter, a11y Speak, Ajax, and many more (especially when viewing a page that uses the block-based editor!).

There are so many utilities in there, it is impossible to document them all, so we are going to pick out a few of the more important and useful ones. At the time of this writing, there is no comprehensive documentation on most of these helpers. Your best bet to learn more today is to open your browser console in Inspector, type `console.log(wp)`, and start poking around to see what is available.

a11y Speak

Accessibility and JavaScript sometimes do not play nicely together. As the web becomes more interactive—more like an application than a page—the need to announce changes in page content has arisen.

`wp.a11y.speak()` exists as the de facto way in WordPress to tell the user, using JavaScript, that the web page content has changed. Information is available at make.wordpress.org/accessibility/handbook/markup/wp-a11y-speak.

Escaping

`wp.escapeHtml` is a polyfill that provides helper functions for escaping quotation marks, ampersands, less-than symbols, HTML, and entire attribute values. As you start to write more React-based JavaScript later, the need to control how your code is escaped will become an important security measure.

i18n

`wp.i18n` is the utility you will use to localize the strings that are embedded inside your React-based JavaScript files. You can now use many of the GetText-style API function calls inside your JavaScript. We will be covering the better known PHP equivalents in [Chapter 11](#).

This is the future of how you will be able to make your JavaScript code ready to be translated into every other language other than English. If you are going to include raw text strings in your JavaScript, we strongly encourage you to use these functions going forward to ensure that it can be localized.

Heartbeat

`wp.heartbeat` is a series of function calls used to manage how routine listeners are enqueued on the current page. As the needs of your application become increasingly complex, you may have things that need to be processed in the background, or you may have application states that need to be checked every few seconds. You would use the Heartbeat API to register how often you need to check for changes and listen to it for a response.

POLYFILLS

Since WordPress 5.0, polyfills exist as a way for WordPress to come bundled with fallback support for many of the JavaScript libraries that are considered to be required going forward for modern web development, in the event that something is missing or a browser does not meet certain requirements.

One example of a polyfill is React, which is a JavaScript library developed by Facebook to assist with rapidly developing reusable and interactive user interfaces. React is used extensively inside Gutenberg, as are a few dozen other of its dependencies.

You may never need to interact with these scripts directly, but it's important to understand what they are and why they exist so that when you start developing your own React components later, you know that React already exists for you here. We will be teaching you all about React in [Chapter 7](#), which covers Gutenberg!

YOUR CUSTOM SCRIPTS

When you are writing advanced WordPress plugins, much of your time will be spent architecting, planning, and researching whether anyone has already solved the same problem that you are going to try to solve yourself. If you're lucky, something already exists, and you will be able to save some time by including their work in with yours. Otherwise, you may be on your own.

The custom JavaScript you write should be broken up into pieces much the same way that PHP files are separated out logically. This is your plugin, and

you should keep your code organized and clean so that it is easy to pick up and work on later, maybe after you've forgotten a lot of what exactly is inside it today. There are dozens of open source tools available for compiling and transpiling your JavaScript, and you should feel comfortable using whichever ones meet your needs.

We recommend plugins have an `/assets` directory, with a `/js` subdirectory where all of the plugins JavaScript can live, all broken out and down into separate files as needed to maintain order within your plugin. You can always minify and compile them together later, but you can't always break everything all apart into smaller more easily managed pieces.

If you are bundling third-party libraries, create another subdirectory for them named `/vendor` underneath your `/js` directory. It is commonly understood that vendor code is code you've incorporated into your project that is outside your direct control but that your own code relies on to get the job done.

- `/your-plugin/assets/`
- `/your-plugin/assets/js/`
- `/your-plugin/assets/js/vendor/`

jQuery

jQuery is the world's most popular JavaScript framework. It is used by more than 80 percent of the top million sites followed by Quantcast (source: trends.builtwith.com/javascript) and is used by many different WordPress admin area pages.

There are many books on jQuery available, and since this is a WordPress book, we aren't going to go into too much detail here.

Benefits of Using jQuery

What makes jQuery such a great library and the reasons why it comes with WordPress are among the following:

- The minified and gzipped library is only 24KB.

- It uses a quick and terse syntax for faster developing.
- It is completely cross-browser compatible.
- It supports all CSS selectors.
- It makes events, DOM manipulation and traversing, and animations and effects all super easy.
- It makes Ajax simpler, easily reading JSON and XML.
- It has great documentation at docs.jquery.com.

It is possible to use any other library you'd like, but because WordPress uses jQuery for its own internal purposes, it's smart for beginners because there are many different examples already in the code base to use for your own starting-off points.

jQuery Crash Course

The scope of this section is not to teach you how to master this powerful JavaScript library in a few minutes but to give you some basis to read on without being completely lost, as well as WordPress-specific information.

The jQuery Object

In vanilla JavaScript, you may have written code like this:

```
document.getElementById('container').getElementsByTagName('a')
```

to select elements that your CSS would call `#container a`. With jQuery, you can now simply write the following:

```
jQuery('#container a')
```

Syntax and Chaining

jQuery methods can be chained together, which will return the jQuery object, as you will see in the following short practical example.

Create a minimalist HTML content that includes the latest jQuery script from the official website.

```

<html>
<head>
<script src='http://code.jquery.com/jquery.js'></script>
<title>Quick jQuery example</title>
</head>
<body>
<p class="target">click on me!</p>
<p class="target">click on me!</p>
<p class="target">click on me!</p>
</body>
</html>

```

Now, right before the closing `</body>` tag, insert a jQuery snippet to add a background and a border to each paragraph, and that, when clicked, changes the background color while shrinking it for two seconds before making it disappear.

```

<script>
jQuery('p.target')
  .css( { background: '#eef', border: '1px solid red' } )
  .click(function(){
    jQuery (this)
      .css('background', '#aaf')
      .animate(
        { width: '300px', borderWidth: '30px',
marginLeft: '100px' },
        2000,
        function(){
          jQuery (this).fadeOut();
        }
      );
  });
</script>

```

If you dissect this compact snippet, you can see the main structure.

```
jQuery ('p.target').css( ).click( function(){ } );
```

This applies some styling to the selected paragraph and then defines the behavior when the event `click` occurs on this element. Chaining enables a method to return an object itself as a result, reducing usage of temporary variables and enabling a compact syntax.

Similarly, within the function defining the `click` behavior, you can see several methods applied to the `jQuery(this)` object, referencing the current

jQuery object instantiated by the initial `jQuery('p.target')`.

What you have now is three independently animated paragraph blocks!

No-Conflict Mode in WordPress

To enable coexistence with other libraries, jQuery has a no-conflict mode, activated by default within WordPress, which prevents using the dollar sign (`$`) directly to call on jQuery itself. The result of this is that if you port existing jQuery code to a WordPress environment, you need to use one of these solutions:

- Write `jQuery()` instead of each `$(())`.
- Use a jQuery wrapper.

To illustrate this, consider the following initial jQuery code you would need to port into a WordPress no-conflict environment:

```
$('.something').each( function(){
    $(this).addClass( 'stuff' );
});
$.data( document.body, 'foo', 1337 );
```

The first option would give the following result:

```
jQuery('.something').each( function(){
    jQuery(this).addClass( 'stuff' );
});
jQuery.data( document.body, 'foo', 1337 );
```

The second option would give the following result:

```
// jQuery noConflict wrapper:
(function($) {
    // $() will work here
    $('.something').each( function(){
        $(this).addClass( 'stuff' );
    });
    $.data( document.body, 'foo', 1337 );
})(jQuery);
```

Both solutions are programmatically equal, but using a no-conflict wrapper will enable you to more conveniently and easily use existing code without

having to replace each \$ with a longer jQuery.

Launching Code on Document Ready

A frequent requirement in JavaScript is to make sure that elements in a page load before you can do something with them. Here is a snippet you may have used before:

```
window.onload = function(){
    /* do something */
}
```

This ancient technique has two weaknesses:

- Another script can easily overwrite the `window.onload` definition with its own function.
- The `onload` event in JavaScript waits for everything to be fully loaded before executing, including images, banner ads, external widgets, and so on.

With jQuery, you get a much better solution:

```
jQuery(document).ready( function(){
    /* do something */
});
```

Now, as soon as the DOM hierarchy has been fully constructed, the document “ready” event triggers. This happens before images load, before ads are shown, so the user experience is much smoother and faster.

You can combine the document-ready function with a jQuery `noConflict()` wrapper, like the following:

```
jQuery(document).ready(function($) {
    // $() will work as an alias for jQuery() inside of this
    // function
});
```

Using this technique, you can use the `$()` syntax and be sure that you do not reference a DOM element that has not been rendered by the browser yet.

Ajax

Ajax is a web development technique that enables a page to retrieve data asynchronously and to update parts of the page without the need to reload the entire thing all at once. The word originally was an acronym for Asynchronous JavaScript And XML, but the use of XML is not actually mandatory.

Ajax is not a technology or a programming language but rather a group of technologies that make up a methodology. It involves client-side scripts such as JavaScript and server-side scripts such as PHP that output content in HTML, CSS, XML, and JSON—or pretty much anything else!

Here is an extremely primitive Ajax example:

```
<!DOCTYPE html>
<html>
<body>
<script>
function loadExample() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("demo").innerHTML =
this.responseText;
        }
    };
    xhttp.open("GET", "example.com", true);
    xhttp.send();
}
</script>

<div id="demo">
    <h2> AJAX Will remove this </h2>
    <button type="button" onclick="loadExample()">Click to
Change</button>
</div>

</body>
</html>
```

You can learn more about Ajax in WordPress by reading the developer documentation at developer.wordpress.org/plugins/javascript/ajax/.

BACKBONE/underscore

Backbone and Underscore were brought into WordPress when the Media Library was redesigned in 2011. They are largely intended to complement jQuery, and therefore we do not recommend investing too much time into learning these technologies, as React is generally considered the modern approach for the foreseeable future. They are worth mentioning here because nearly everything in the Media Library uses them, which is cleanly coded and fully documented. If you are curious to learn more about them, please refer to the Media Library code itself.

Lodash is also included in WordPress and in many ways is the successor to Underscore, much in the same way that using React will mean you will not really need to use Backbone.

Please visit backbonejs.org and underscorejs.org for more information.

REACT

React was invented by engineers at Facebook and is the new star of the JavaScript show. Not only is it bundled with WordPress, but its popularity has eclipsed that of other modern competitors like Angular and Vue. Its licensing was originally incompatible with the GPL license that WordPress uses, making it a complete nonoption at its inception. The web development community rallied to get Facebook to reconsider its licensing and it did, so here we are!

React by itself is a lovely and simple piece of software. Using it in WordPress today, though, does require a little bit of rule bending and head scratching. This is because React is new and designed to be used alongside a bevy of new JavaScript libraries and tools, but WordPress is a legacy piece of software with many years' worth of compromises that have been made over time.

How to best use React in WordPress has been iterated on and reinvented and changed many times since WordPress 5.0 was released. The dust is starting to settle, so we are going to teach you what is currently considered to be the best way to blend them both together in your own plugins.

Much like how jQuery UI complemented jQuery, libraries such as Babel, Redux, Moment, webpack, React DOM, and others exist in WordPress to complement React and to help make things like the block-based editor—Gutenberg—possible.

Rather than try to deliver a complete crash course in React here, we are going to jump straight into applying everything we've learned so far in the next chapter. React requires a totally different (and very deep) dive into JavaScript concepts that are not really used anywhere else in WordPress. Get ready, because the next chapter is going to be intense!

SUMMARY

JavaScript now makes up around 30 percent of the WordPress code base, and there are no signs of that number getting smaller anytime soon. Just like with PHP, there is now an endless amount of material available that will take you on super-deep dives into just about every different aspect of it.

This chapter has focused solely on the WordPress-specific things that most other JavaScript texts would not think to include. You will be writing more and more JavaScript in your professional WordPress plugins, and there is no avoiding it anymore. [Chapter 7](#) will continue your JavaScript education by fully immersing you in the wonderful world of Gutenberg blocks!

7

Blocks and Gutenberg

WHAT'S IN THIS CHAPTER?

- Introducing Gutenberg
- Using blocks
- Using WP-CLI

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are at

www.wiley.com/go/prowordpressdev2e on the Downloads tab.

WordPress is known for being the best software for publishing to the web, and a major reason for that is its inclusion of an absolutely brilliant visual editor, known as TinyMCE. TinyMCE is without a doubt the world's most popular open source, web-based, what-you-see-is-what-you-get (WYSIWYG) editor. It has been the paper on which millions of authors have inked their online publications since the earliest versions of WordPress, and many people have switched to WordPress over the years because of it.

WHAT IS GUTENBERG?

While WordPress and TinyMCE were gaining in popularity, several smaller, competing editors were developed and found much success, even surpassing TinyMCE's abilities. These other editors provided web tools to empower authors to simultaneously write great content and make it look like something that had been handcrafted by artisans. WordPress could not do that on its own. It required shortcodes, widgets, and themes—sometimes several of them together or combined—to accomplish what services like Squarespace and Wix were doing without any additional configuration or complexity.

So, the WordPress project decided it was time for a change to its tried and trusty publishing experience—a change that would blur the lines between web publishing and web design. WordPress decided to implement modern coding techniques and approaches that had otherwise been far outside the realm of what WordPress had been known for up until then.

Called Gutenberg and released in WordPress version 5.0, this new approach to online publishing would shy away from simply mimicking a word processor and instead become a “block-based editor,” with each block having its own customizable set of attributes and allowing for much simpler manipulation of more complex types of media and components. Instead of needing to write custom code and relying on plugins to provide their own difficult-to-discover and bespoke shortcode tags, you can use Gutenberg blocks. These *blocks* are organized into categories and provide a consistent and familiar set of interface elements that users can use to quickly and easily create relatively sophisticated page layouts alongside their words.

This new editor is named after Johannes Gutenberg, the German inventor known for introducing movable type printing to Europe in the 1430s and kickstarting the beginning of the printing revolution. Johannes' multiple contributions to the printing and publishing process are credited for being the catalyst to the knowledge-based economy we live in today, helping all humans around the world share their words more freely and easily than had ever been previously possible. Needless to say, the name is deeply rooted in the importance of publishing words.

When this new editing experience was finally shipped to the world, Gutenberg was not very well received. Unlike Johannes' movable type, not everyone immediately saw the benefits of such a drastic shift in publishing priorities and interfaces. In fact, a plugin that was released alongside WordPress 5.0 that brings back the Classic Editor continues to be one of the most popularly installed plugins. It even spawned a fork of WordPress called ClassicPress that garnered early support because it did not use Gutenberg. However, as Gutenberg has improved and as WordPress charges forward with it, developer momentum is increasing.

The team of talented folks currently creating Gutenberg estimate that it is about 10 percent complete. The next two major milestones to hit include merging with the Customizer and ultimately including full-site

customization. Much has already changed since Gutenberg was first released, and while folks have already had a difficult time keeping up with the pace of development, it is unlikely to slow down for the next few years.

It is largely understood in the WordPress community that this block-based editing experience is the future of WordPress. Ready or not, if you are (or are aspiring to be) a WordPress professional, you need to know how to use it and how to extend it because you will eventually need to fully embrace and support it. As of WordPress 5.3, all of the built-in widgets and shortcodes have block equivalents, so you could theoretically create a complete site using only blocks and nothing else. That really is the goal for future versions of WordPress, and we are currently seeing the beginning of that future take shape.

This chapter is focused on the block-based editing experience known as Gutenberg. It attempts to explain in relatively simple terms several new concepts that even many seasoned WordPress professionals are likely to be unfamiliar with. It ends with you building your own dynamic block in what is currently the most professional way possible.

GUTENBERG LINKS

If you'd like to keep up with what's going on day-to-day in Gutenberg, here are a few links that are important to keep up with as Gutenberg evolves over the next few years:

<https://github.com/WordPress/gutenberg>

<https://wordpress.org/plugins/gutenberg>

<https://wordpress.org/gutenberg>

<https://developer.wordpress.org/block-editor>

TOURING GUTENBERG

To best work with Gutenberg, you first need to understand what it looks like. In this section, we will draw your attention to specific areas of Gutenberg so that they are familiar to you later when you need to start interacting with them in your own custom code.

By now, you should have a comfortable development environment running where you can freely experiment. If for whatever reason you don't, you are in luck because the WordPress project hosts its own interactive Gutenberg demonstration at <https://wordpress.org/gutenberg>.

Before we get too far ahead, let's take a look at what the Classic Editor looks like ([Figure 7-1](#)).

If you've ever used WordPress before, this interface should probably look pretty familiar. The prominent title, columns, meta boxes, strong borders, and a generous amount of padding around everything are clues we are looking at the Classic Editor.

Compare the Classic Editor interface to [Figure 7-2](#), which shows what the block-based Gutenberg editor looks like.

At first glance, there are a lot of similarities, but as you start to look more closely, you will start to notice quite a few major differences. There is no formatting toolbar. The Add Media button is completely missing. All of the individual meta box sections are collapsed instead of open. Finally, there are a bunch of totally new icons and symbols to learn about.

Starting at the top left of the toolbar, hover over the + icon and click it, and you'll see the Block Library menu, as shown in [Figure 7-3](#).

Here is your first look into all of the blocks that come packaged with WordPress. Anytime you see a + icon (in the toolbar or in the content area), clicking it will reveal some version of this menu and expose the Block Library. By the end of this chapter, you'll have added your own block into the Block Library with your own custom functionality!

Collapsing the Most Used section will reveal all the various categories that all the different blocks are registered with. We aren't going to go through all of them together, but you can quickly see what they are in [Figure 7-4](#).

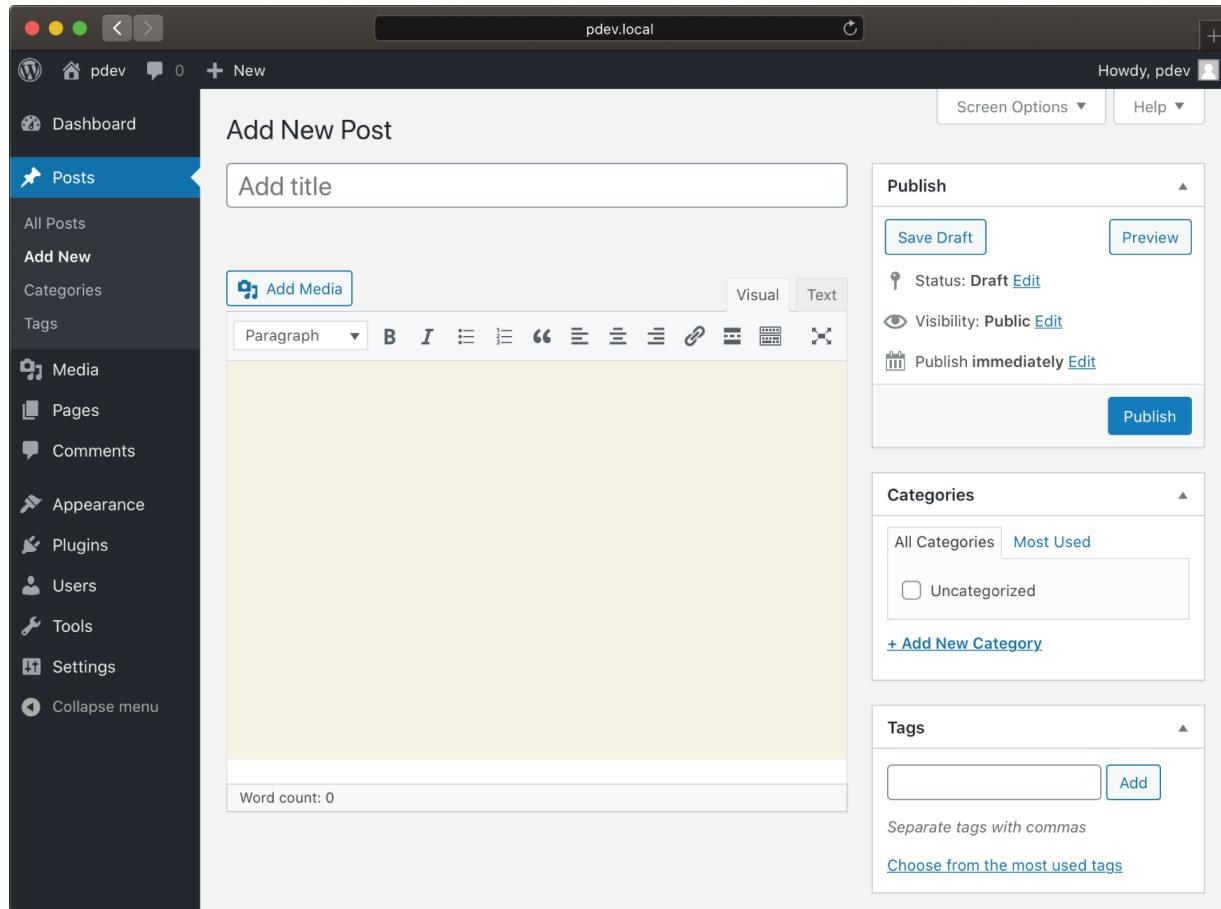


FIGURE 7-1: Classic Editor, not covered in this chapter

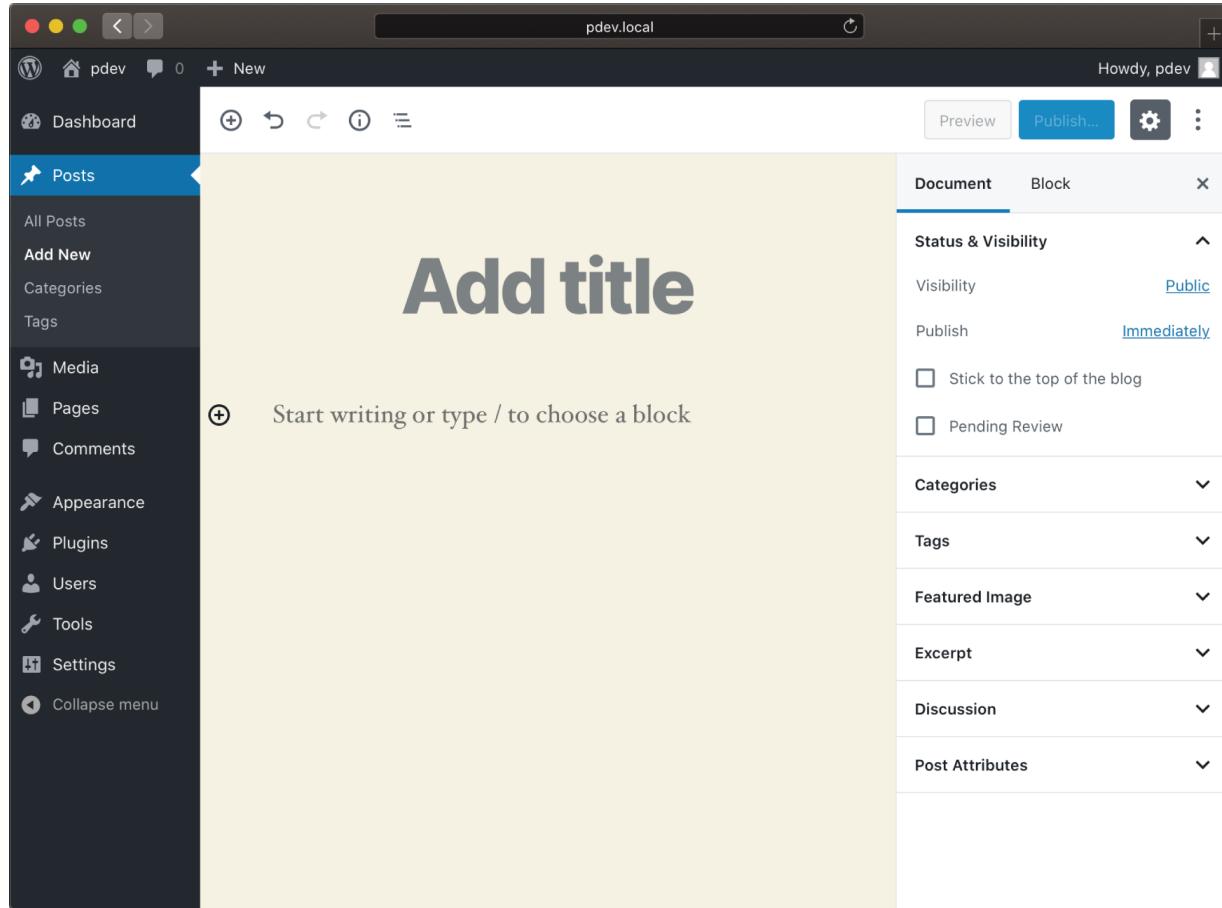
Next to the + icon in the toolbar are Undo and Redo buttons, a Structure button that shows some quick statistics (such as word count and block count), and a Block Navigation button that reveals a neat hierarchy view of the entire document (this will make more sense once you've composed a complex post or page).

On the top-right side, you'll notice the same Preview and Publish buttons you've always had, only here they're broken out from the sidebar or meta box and now are always available in the formatting toolbar. Pretty convenient!

The gear icon will show and hide the sidebar, which is a terribly generic name for what is ultimately a document and block inspector interface. In this sidebar is where you can find all of the various attributes for the entire post or for a single block if you happen to have focused on one at the time.

Next to the gear icon is a vertical ellipsis icon; clicking it will reveal some view options for the entire editing experience, as shown in [Figure 7-5](#).

Choosing view settings is ultimately a matter of user preference, and you should feel comfortable clicking through all of these and checking them out. These settings will not impact any of what we are going to be working on going forward.



[FIGURE 7-2](#): Gutenberg

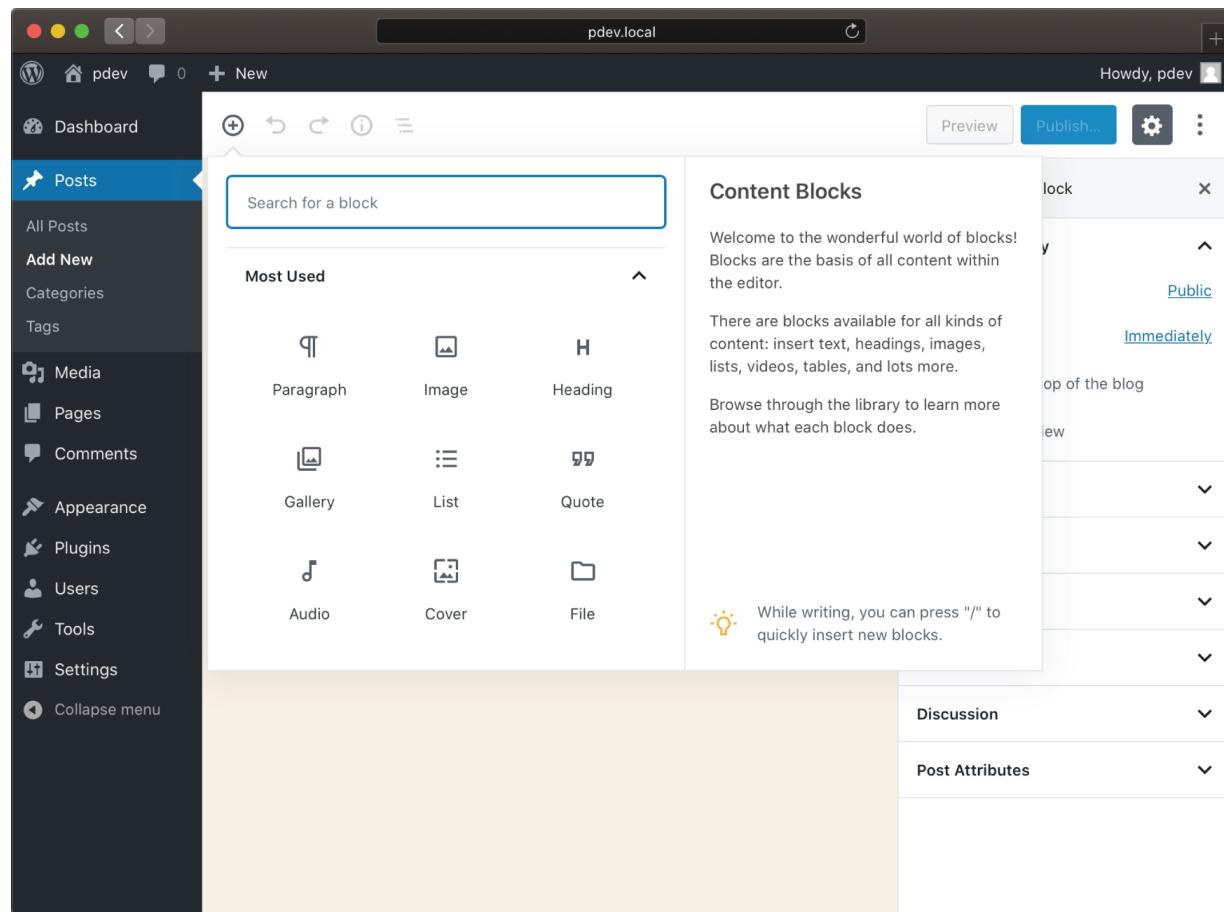
A lot of folks seem to like having a persistent formatting toolbar, as it reminds them of the Classic Editor. The reason we like it is because it eliminates a lot of visual noise when navigating around posts with many blocks, so we recommend giving it a try.

[Figure 7-6](#) shows what it's like to edit the "Hello world!" blog post. You can see in the sidebar that the Document menu is showing various subsections that expose certain document properties, many of which should look familiar from the Classic Editor.

[Figure 7-7](#) shows the sidebar's Block menu, and the first paragraph block is selected in the post content on the left, giving us our first look at the revamped formatting toolbar.

The formatting toolbar changes depending on what kind of block is selected and what properties it has available to it. Certain types of blocks can easily be converted into other types of blocks, while others can only be what they are. You can also use the formatting toolbar to apply formatting to specific words within the block content.

The sidebar's Block menu briefly describes the block type, and you will see all of the properties for that block that can be changed and applied, all while viewing the results visually and in real time in the content area.



[FIGURE 7-3](#): Block Library menu

As you have seen during this brief tour, a lot has changed, and a lot has stayed the same. Gutenberg is similar enough to look familiar but foreign enough to feel just a little bit intimidating—maybe even a little bit

confusing at times—especially if you have 10 years' worth of muscle memory behind you with the Classic Editor!

GLOSSARY

Here we've defined a few of the most commonly used terms that are unique to block-based editing. In the event you forget what something is, this can serve as a handy cheat sheet to come back to.

Block: Anything from a simple single paragraph HTML tag to a complex dynamically rendered set of HTML elements

Block Library: The place where all registered blocks can be found, searched, and selected

Block category: A group or collection of blocks to make them easier to find in the Block Library

Content area: The large blank canvas where words are typed and blocks are inserted

Formatting toolbar: The horizontal bar that provides convenient access to apply different styles

Sidebar: Found on the opposite side of the screen as the admin menu, a document and block inspector

Sidebar menu: Part of the sidebar, responsible for different aspects of the overall document depending on whichever block currently has focus

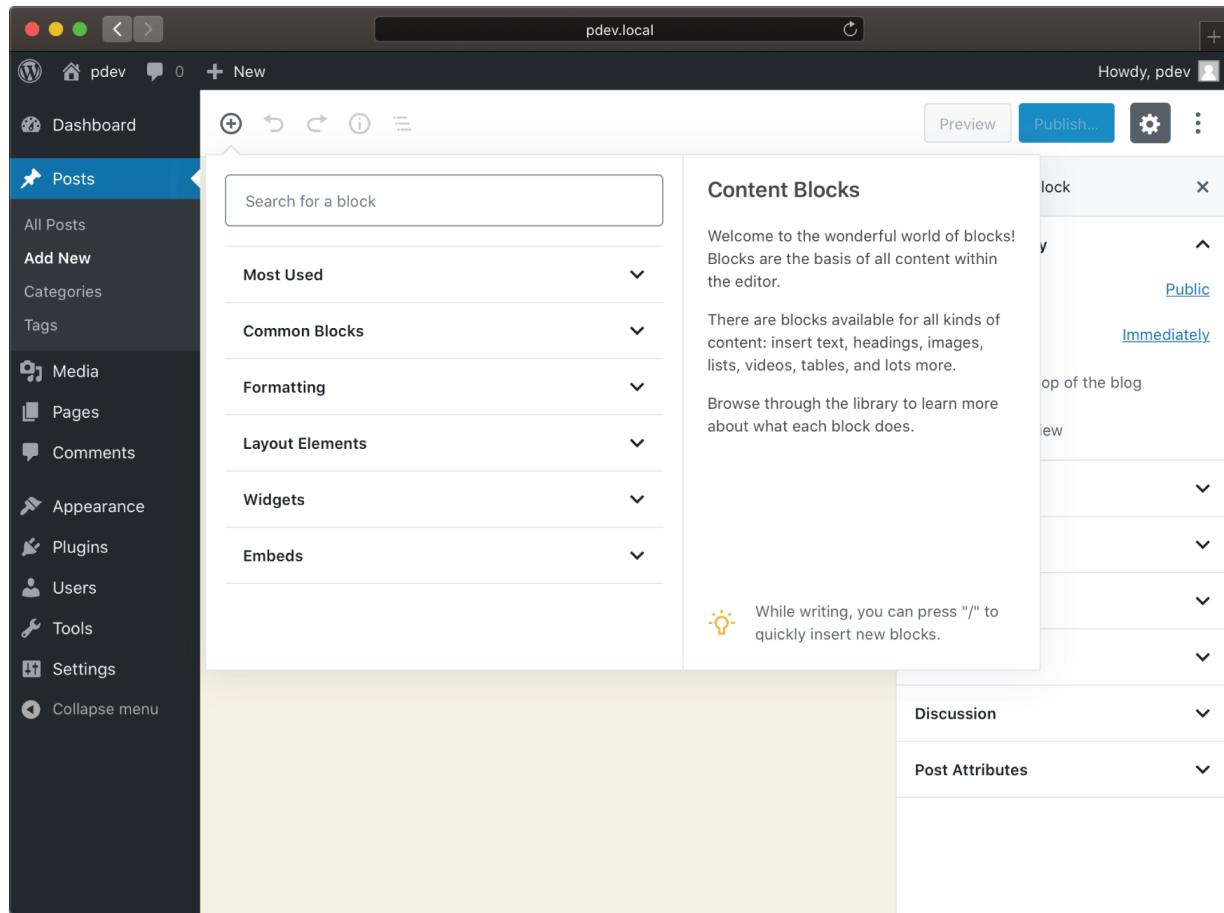


FIGURE 7-4: Categories of blocks

PRACTICAL EXAMPLES

A few of the more popular WordPress plugins available in the [WordPress.org Plugin Directory](https://wordpress.org/plugins/) have taken the initiative to introduce their own custom blocks to support this new Gutenberg editor.

In this section, we will show you a few of those plugins so you can get some inspiration from what other professionals are already doing with it.

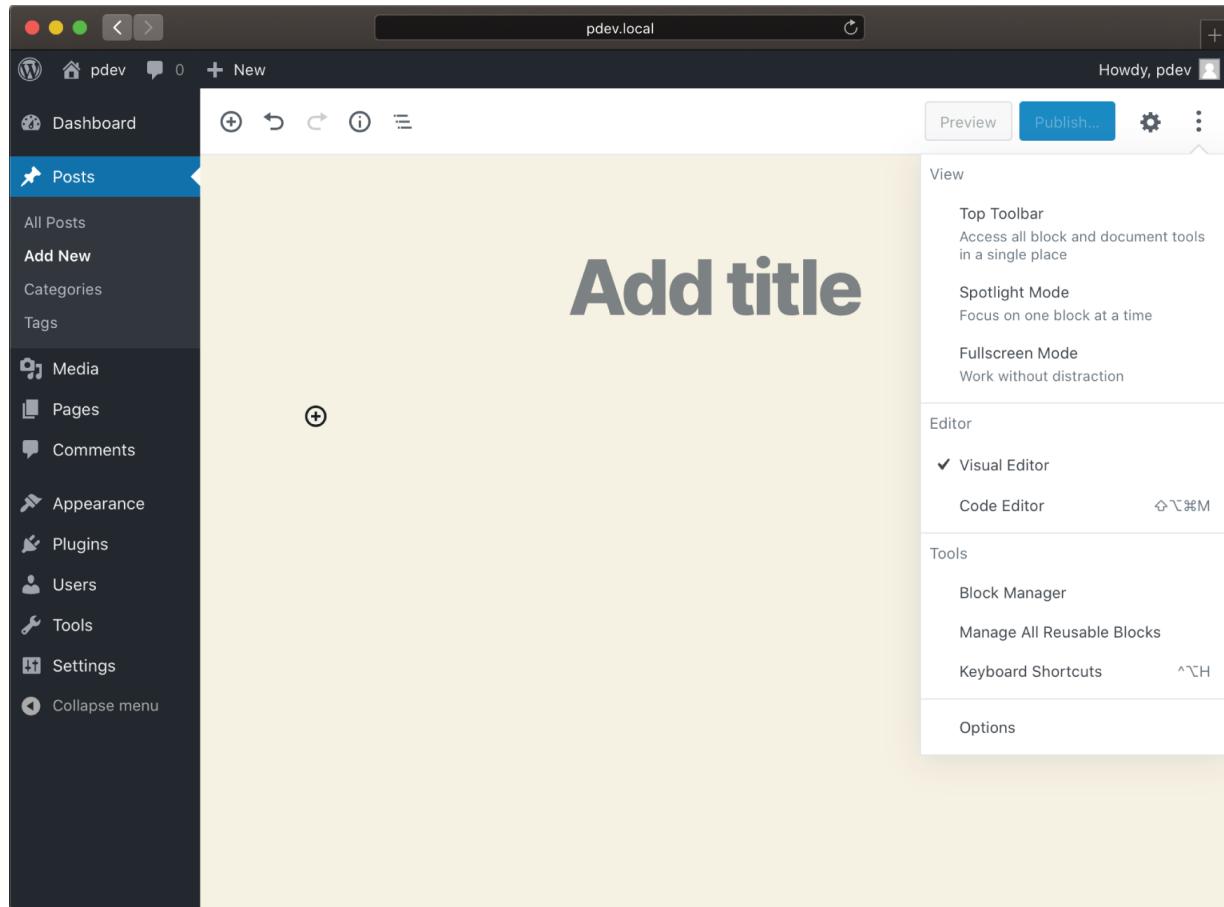


FIGURE 7-5: View options

WooCommerce

With more than 5 million installations and a team of more than 100 people working on it, the folks at Automattic were able to crank out a bunch of really useful blocks for WooCommerce right away, as shown in [Figure 7-8](#).

WooCommerce comes with more than a dozen custom blocks for users to pick from to help them lay out their product pages. WooCommerce uses its signature purple for its iconography, which is available as an icon attribute when registering blocks.

[Figure 7-9](#) shows off the Newest Products block, and as you can see in the sidebar, it comes complete with quite a few editable block properties.

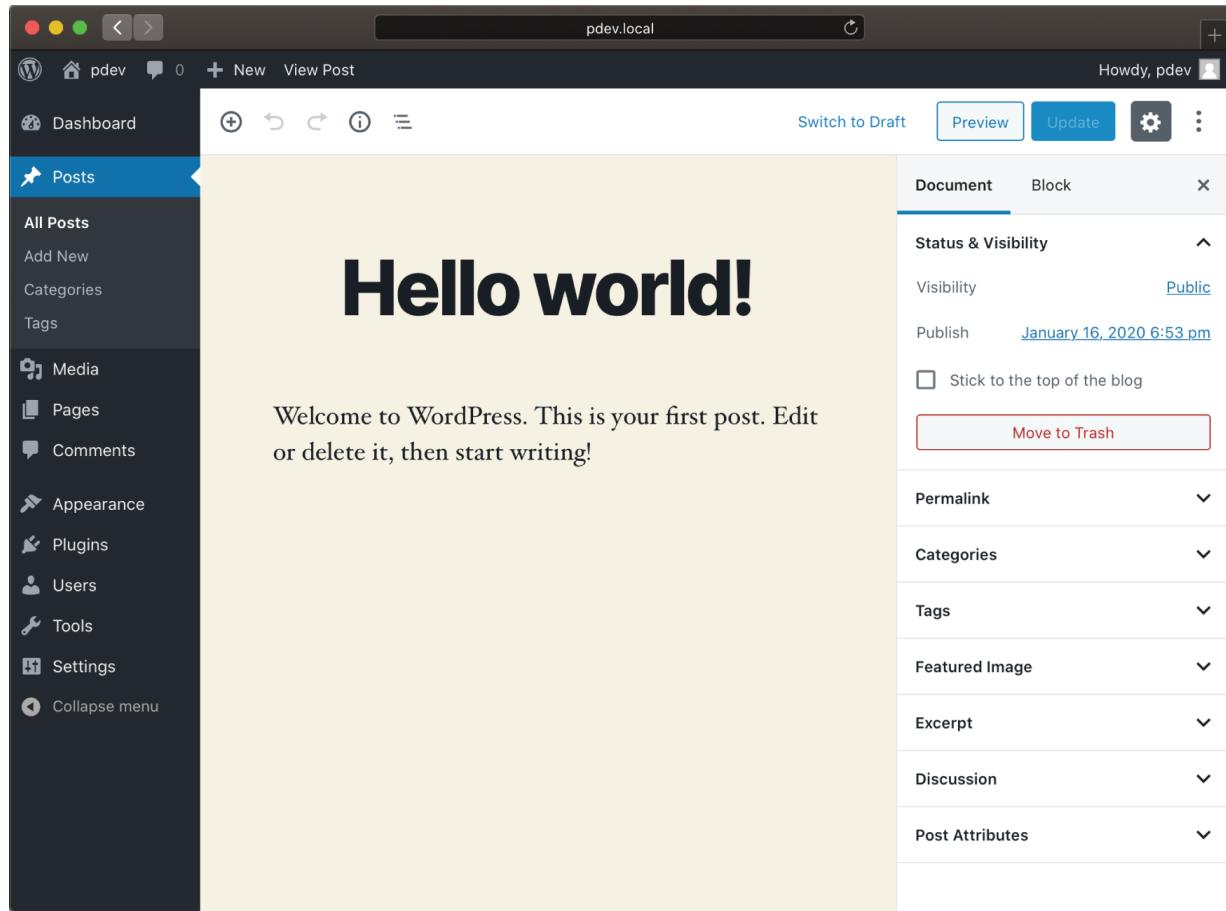


FIGURE 7-6: Sidebar's Document menu

The Events Calendar

Modern Tribe took a completely different approach to supporting the block-based editor with its plugin called Events Calendar. It is completely owning the “add new event” experience on its own admin area pages, as shown in [Figure 7-10](#).

These custom blocks are available only on Event Calendar's Add New page, and the content area comes preloaded with all of the blocks you need to set up an event; all you as a user need to do is fill in some fields and hit Publish to call it done. Pretty neat!

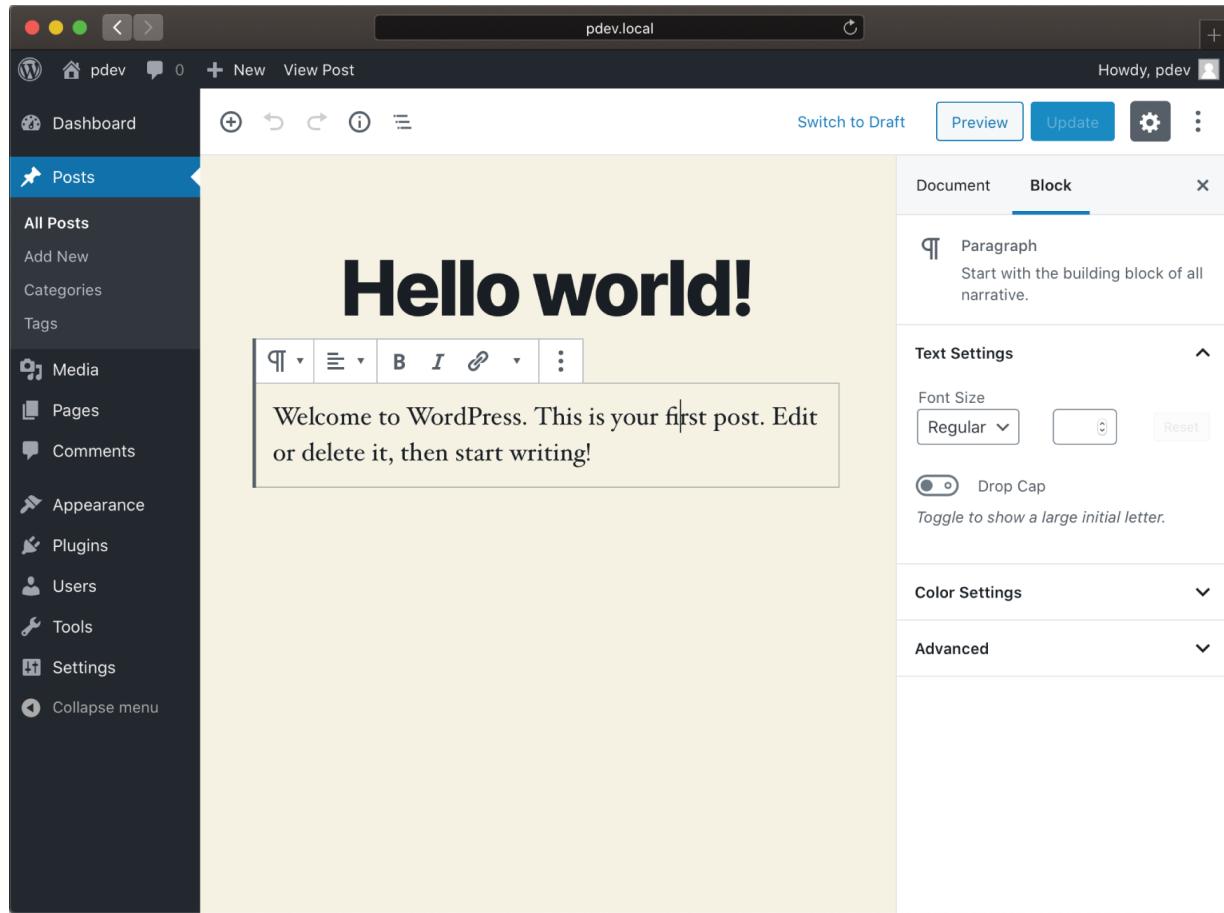


FIGURE 7-7: Sidebar's Block menu and formatting toolbar

Event Calendar does not use custom icon colors like WooCommerce did. Instead, it uses a few of its own custom icons, which have been assigned and used in a number of different, creative ways.

Post Type Switcher

While the Post Type Switcher plugin does not include its own blocks, it does support the block-based Gutenberg editor with a simple but familiar interface. See [Figure 7-11](#).

The Post Type Switcher plugin adds a new Status & Visibility setting to the sidebar's Document menu, allowing for user-facing post types to be switched to other user-facing post types that the current user has adequate capabilities to switch to.

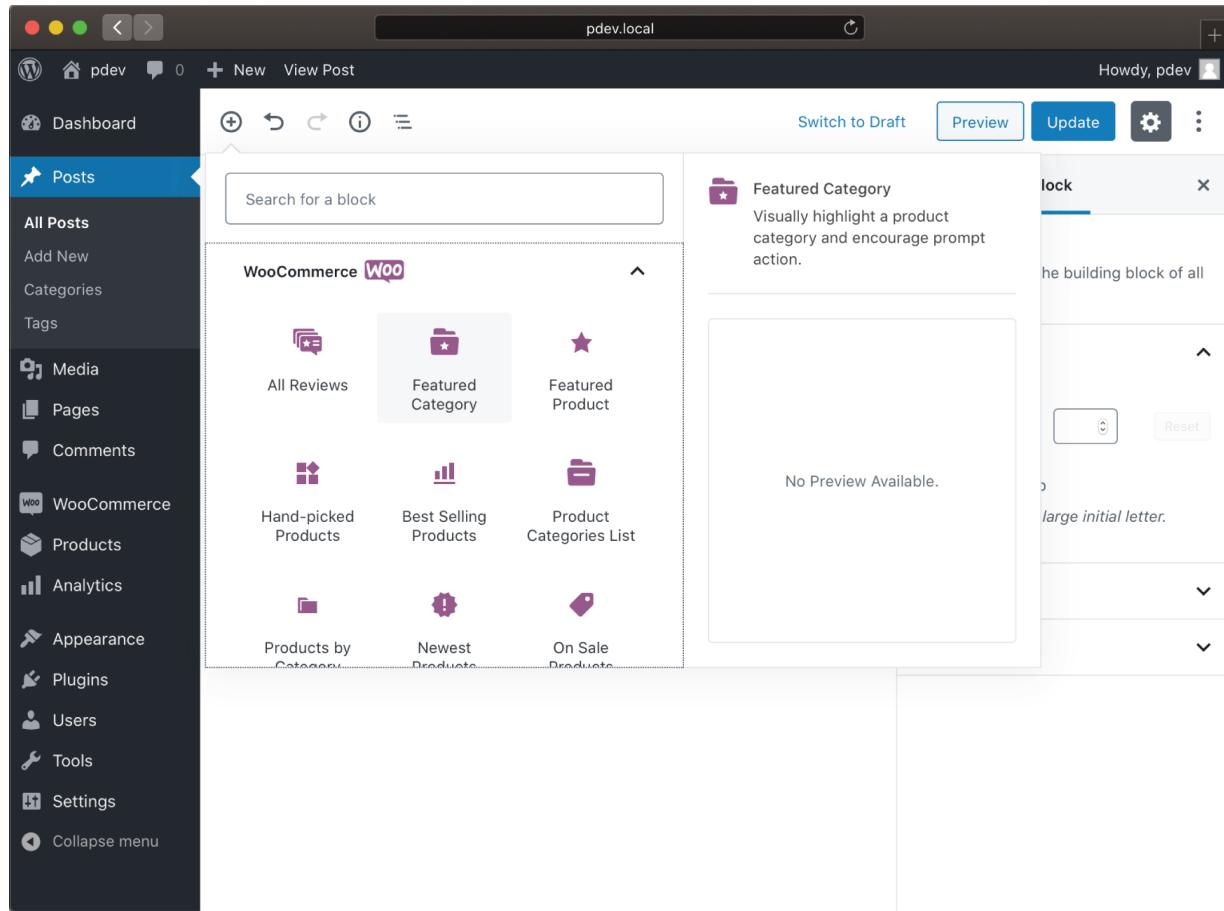


FIGURE 7-8: WooCommerce blocks

By building in support for Gutenberg, this plugin helps maintain user confidence that it is being maintained and kept up-to-date with the latest versions of WordPress.

As a professional plugin developer, keeping your plugins up-to-date with the latest version of WordPress is important. We've talked about it in previous chapters, but it is especially true for major new features like Gutenberg.

TECHNOLOGY STACK OF GUTENBERG

Gutenberg's block-based editing experience is built using nearly 100 percent JavaScript. From start to finish, the only PHP in it is the bare minimum that is necessary to bridge its functionality with WordPress. The rest of the (relatively huge and not getting smaller) codebase is a self-

contained and fully compiled JavaScript web application, which just happens to also run inside WordPress pretty darn well.

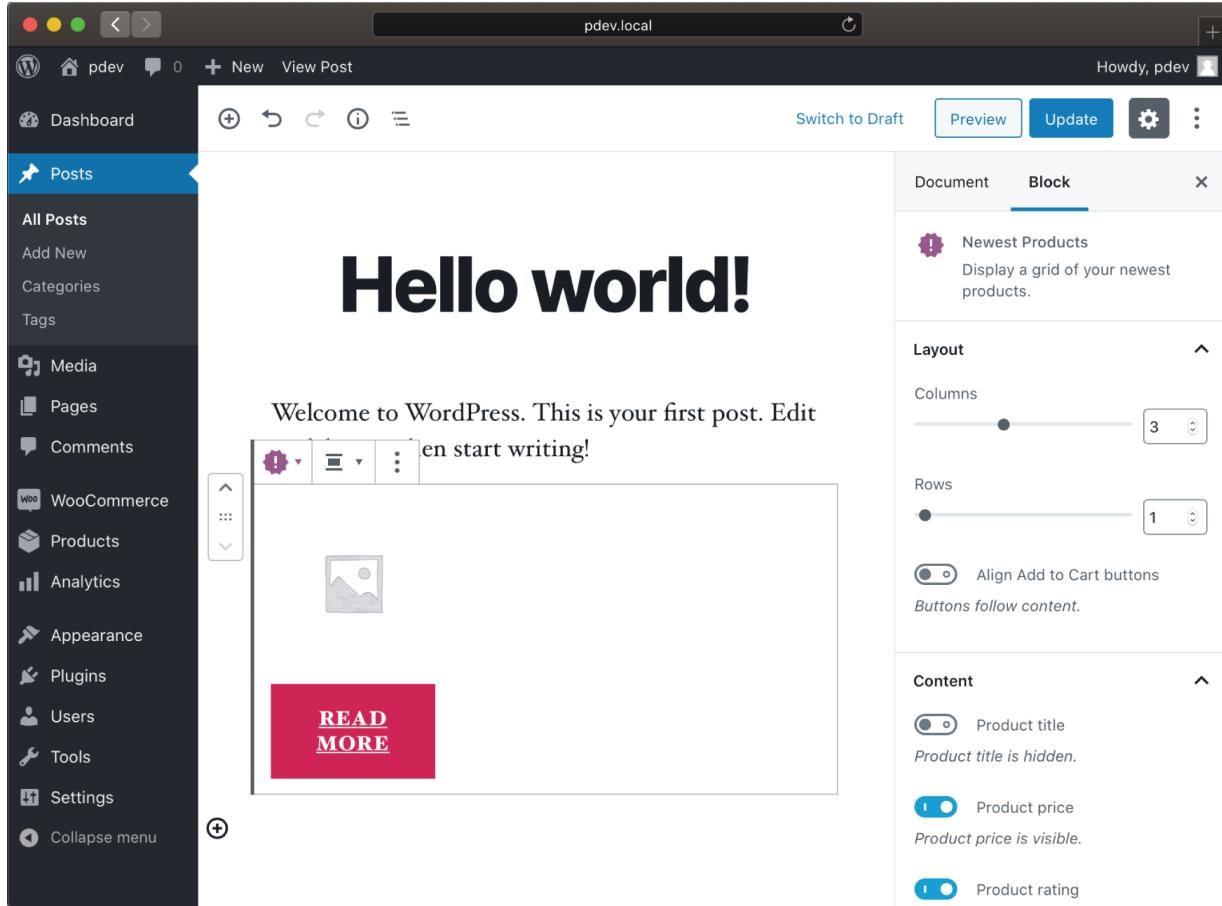


FIGURE 7-9: Newest Products block

The world of modern JavaScript is going to be completely unrecognizable to your average WordPress and PHP developer. In the following sections, we will briefly describe all of the various tools that are necessary for a professional plugin developer to get started making your own custom block.

In addition, it's important to remember that [WordPress.org](https://developer.wordpress.org/block-editor) has a living handbook that will always have up-to-date instructions on how to make custom blocks:

<https://developer.wordpress.org/block-editor>

JavaScript

Alongside HTML and CSS, JavaScript is one of the core technologies of the web, and it allows websites to be interactive rather than static. You've created .js files before and have used some jQuery in [Chapter 6](#), so you've definitely written some JavaScript, but probably nothing like what you are about to write.

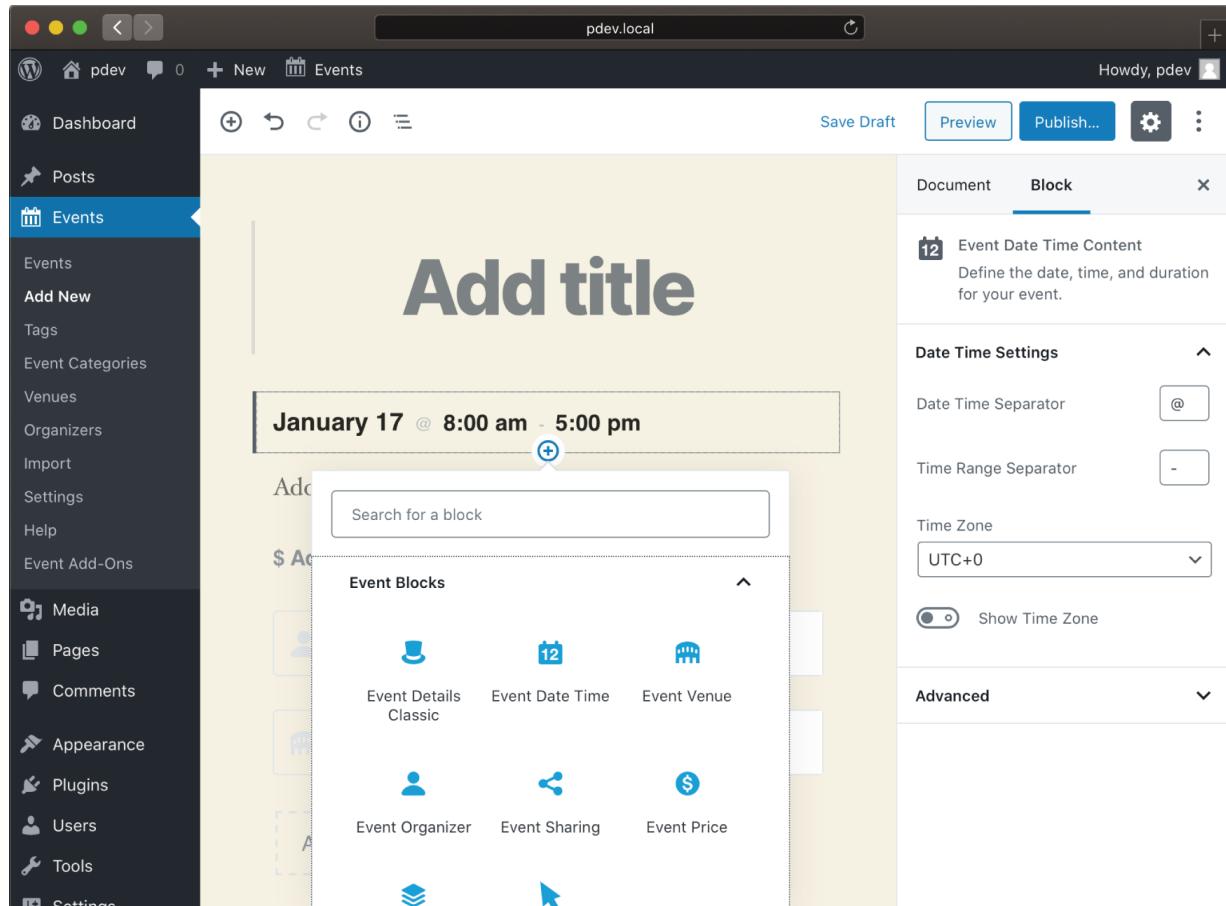


FIGURE 7-10: Event Calendar blocks

Vanilla JavaScript is commonly used to refer to plain JavaScript, or JavaScript that does not rely on any framework (such as jQuery) to run. It is important that you at least be familiar with many of the basic syntactical requirements of vanilla JavaScript (being able to use basic variables, arrays, objects, and so on).

There are dozens of great books available that focus entirely on JavaScript, as well as good online resources. You can learn more about the organizations behind JavaScript and ECMAScript at <http://www.ecma-international.org>.

PHP

Many of the original WordPress core blocks have PHP variants that can be found in the `wp-includes/blocks` directory, though they do have JavaScript that accompanies this PHP. There are several helper functions in `wp-includes/blocks.php`, screen methods such as `is_block_editor()`, and admin area pages written in PHP that exist solely to support the block-based Gutenberg editor.

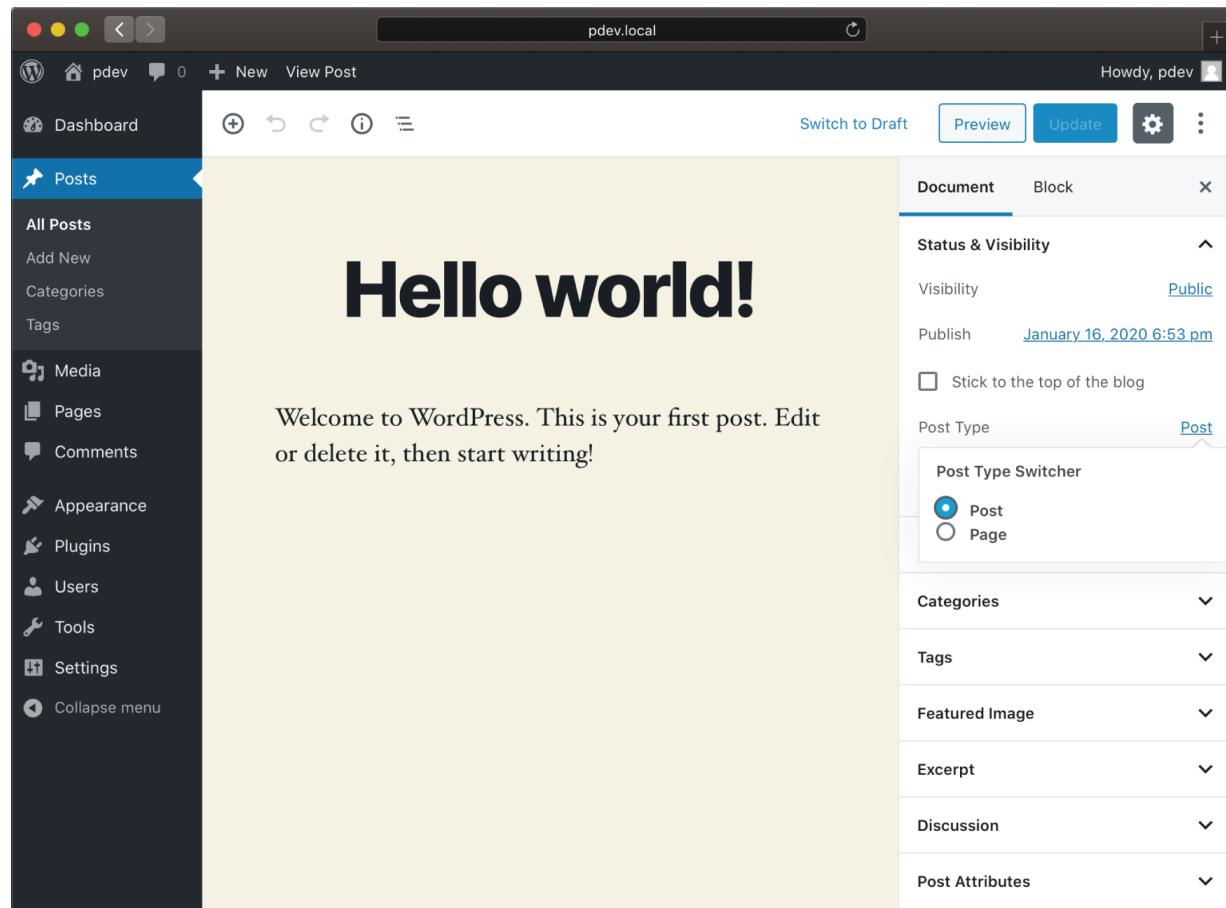


FIGURE 7-11: Post Type Switcher plugin

Depending on the needs of your plugin or application, you may still want to use some hybrid Ajax approach with PHP callbacks to handle any of the server-side validation that might be necessary.

You can learn more about the PHP language, project, and community at <https://www.php.net>.

Node.js

Node.js is a runtime environment, similar to Nginx or Apache, and allows for the creation of web servers and networking tools using JavaScript. Unlike PHP, Node.js requests are nonblocking and can run concurrently or in parallel.

Node.js has a huge selection of modules, including a package manager (npm), to help make managing packages easy. It has governance and many large corporate sponsors. Much of the JavaScript we will write will require Node.js in some fashion.

Read all about Node.js at <https://nodejs.org>.

webpack

webpack is a JavaScript module bundler, taking modules with dependencies and generating static assets representing those modules. It is not JavaScript-specific but will be used for JavaScript and our purposes here.

It also allows for “code splitting” to improve the performance of a bundle of modules and assets.

We'll be using webpack to compile all of our JavaScript into a single file, just like Gutenberg does.

Learn more about webpack at <https://webpack.js.org>.

Babel

Babel is a JavaScript compiler that allows developers to write the most modern syntax version of JavaScript and that converts that code into something that is backward compatible for older browsers and environments.

It includes polyfills for missing JavaScript features and source code transformations to mutate code during the build process and is completely pluggable and debuggable with source maps and more.

We'll be using Babel to write ES6 and JSX (both covered in the following sections) and trust that it will get converted into vanilla JavaScript that all modern browsers can interpret.

React

React is a JavaScript library invented and sponsored by Facebook and is used for building user interfaces that inherently understand how to “react” when source data changes.

It allows us to build encapsulated components that are in complete control of their own states, keeping their logic in JavaScript and keeping their states out of the DOM.

We'll be writing all of our front-end, user-facing interfaces as React components.

JSX

JSX is a syntax extension to JavaScript that is particularly useful with React, as it can be used to quickly produce React “elements.”

Do not make the same mistake we did and confuse this JSX with another JavaScript project named JSX that does something entirely different.

We'll be touching on JSX in React only briefly, so we encourage you to explore it in greater detail on your own at

<https://reactjs.org/docs/introducing-jsx.html>.

ES6

ES6, also known as ECMAScript 2015, is the most widely adopted and recommended flavor of ECMAScript in use today and across this entire technology stack. We'll be writing this style of JavaScript when building our React components.

Just like JavaScript, ECMAScript is primarily used for client-side scripting in the web browser and is also gaining popularity for server-side applications and services because of Node.js.

We recommend visiting this website for more information on ES6:

<http://es6-features.org>.

“HELLO WORLD!” BLOCK

This should probably go without saying by now, but even getting something as simple as your typical “Hello world!” type of block working is going to

require at least a little bit of finesse. This is because we have a few different systems we need to talk to, spanning a few different dependency trees, including a basic user interface.

To get things moving quickly, we're going to write a small but effective plugin to help you make your first custom block.

Create the directory `wp-content/plugins/pdev`.

PHP

We are going to need a way to activate our plugin and enqueue our JavaScript. Type the contents of [Listing 7-1](#) in a file named `pdev.php`.

[LISTING 7-1](#) Enqueueing our JavaScript

```
<?php
/**
 * Plugin Name: PDEV
 * Description: Hello world!
 * Version:      1.0.0
 * Author:       PDEV
 */
add_action( 'enqueue_block_editor_assets', function() {
    wp_enqueue_script(
        'pdev/hello-world',
        plugins_url( 'pdev.build.js', __FILE__ ),
        array( 'wp-blocks', 'wp-element' )
    );
} );
```

JavaScript

We are also going to need register our custom block type so it shows up in the Block Library. Type the contents of [Listing 7-2](#) in a file named `pdev.src.js`.

[LISTING 7-2](#) Registering our block type

```
const { registerBlockType } = wp.blocks;

registerBlockType( 'pdev/hello-world', {
    title:      'Hello world!',
    icon:       'admin-site-alt2',
    category:   'common',

    edit() {
        return <pre>Hello world!</pre>;
    },

    save() {
        return <pre>Hello world!</pre>;
    }
} );
```

webpack

It's way easier to rely on webpack to handle the dependencies that our plugin must have on `wp-blocks` and `wp-element`, so we are going to create a simple configuration file to help us.

We need webpack to know where our source file is, where the build file should be, and what modules and presets we are going to use. In our case, we simply need `env` and `react`.

Type the contents of [Listing 7-3](#) in a file named `webpack.config.js`.

LISTING 7-3 webpack configuration file

```
const path = require('path');

module.exports = {
  entry: "./pdev.src.js",
  mode: "development",

  output: {
    filename: "pdev.build.js",
    path: path.resolve(__dirname, "")
  },

  module: {
    rules: [
      {
        test: /\.m?js$/,
        exclude:
        /(node_modules|bower_components)/,
        use: {
          loader: "babel-loader",
          options: {
            presets: [
              "@babel/preset-env",
              "@babel/preset-react"
            ]
          }
        }
      }
    ]
  }
};
```

Command Line

Using your favorite command-line interface, navigate to the directory we just created, and type the following:

```
npm init
```

You will be prompted to enter a bunch of information, but you can leave everything empty since this is just a simple example.

Next we need to introduce our own “build” command so that npm knows exactly what it is expected to do. We’ll do that by editing the package.json file that the previous command created for use and adding it so that the scripts portion looks like this:

```
"scripts": {  
  "test": "echo \\\"Error: no test specified\\\" && exit 1",  
  "build": "webpack"  
},
```

Next we need to install a few specific Node.js modules using npm (Babel, webpack, and React):

```
npm install --save-dev webpack webpack-cli babel-loader  
@babel/  
  preset-react @babel/core @babel/preset-env
```

Lastly, type the following:

```
npm run build
```

Your terminal should match what is in [Figure 7-12](#), indicating that webpack finished successfully.

```
> npm run build  
> pdev-hello-world@1.0.0 build /Users/jjj/Development/Local/pdev/app/public/wp-content/plugins/pdev  
  webpack  
  
Hash: 52a7df79209919045fd0  
Version: webpack 4.41.5  
Time: 906ms  
Built at: 01/17/2020 5:21:14 AM  
          Asset      Size  Chunks      Chunk Names  
pdev.build.js  4.28 KiB  main  [emitted]  main  
Entrypoint main = pdev.build.js  
[ ./pdev.src.js] 485 bytes {main} [built]
```

FIGURE 7-12: webpack finishing successfully

Activation

Now go activate this plugin and edit (or add) a post. You should now see our “Hello world!” block in the Block Library, as shown in [Figure 7-13](#).

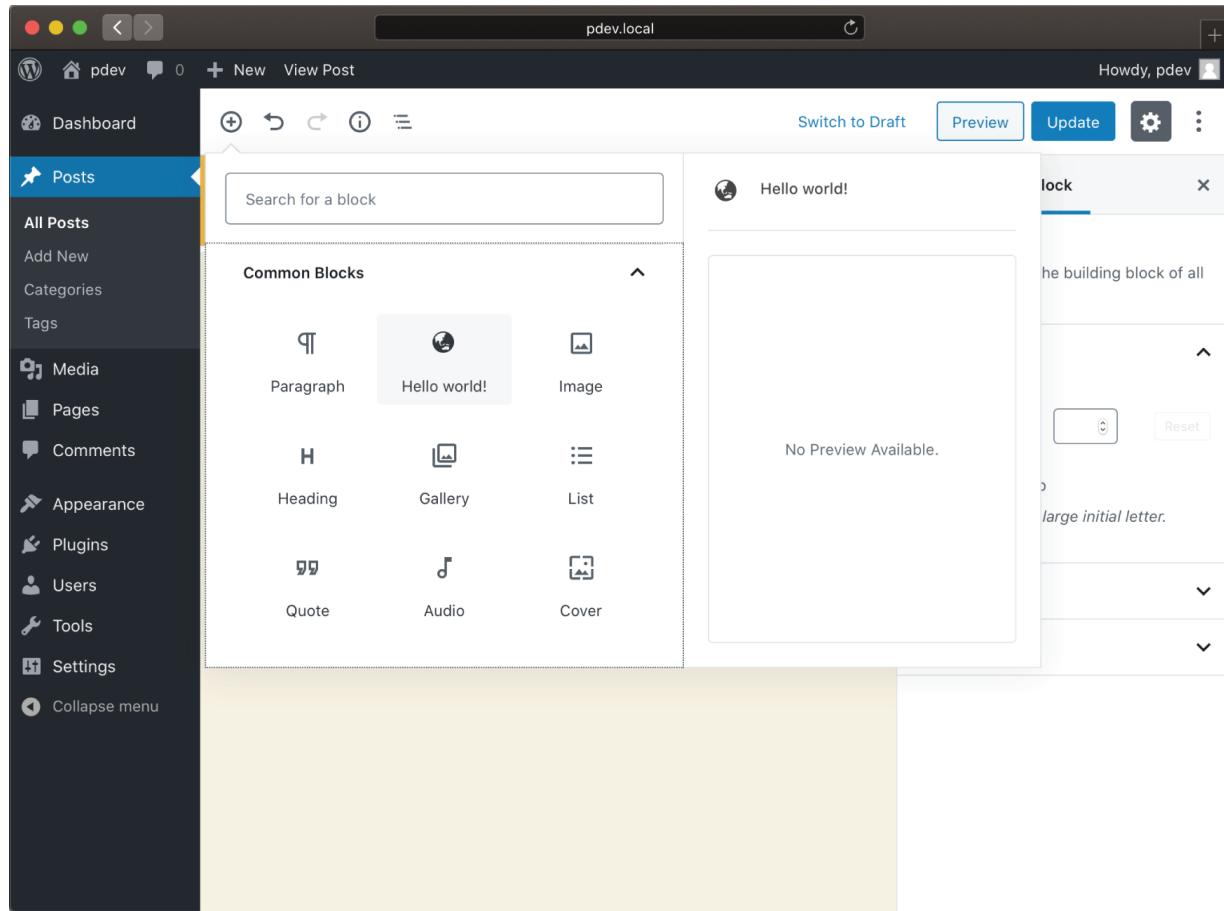


FIGURE 7-13: Our “Hello world!” block in the Block Library

Selecting our block will look like [Figure 7-14](#).

Wrap-Up

Voilà! You've just created your first Gutenberg block. It's simple, but it's also a little bit scary, because you really need to depend on a bunch of new tooling to compile and generate the necessary JavaScript for you.

Like we said earlier, you technically could write your own 100 percent vanilla JavaScript, but once you do, you'll learn just how painful that is to repeat multiple times. Soon you'll be back to looking at webpack and Babel and React all over again, so it really is best to start out using at least some of the tools.

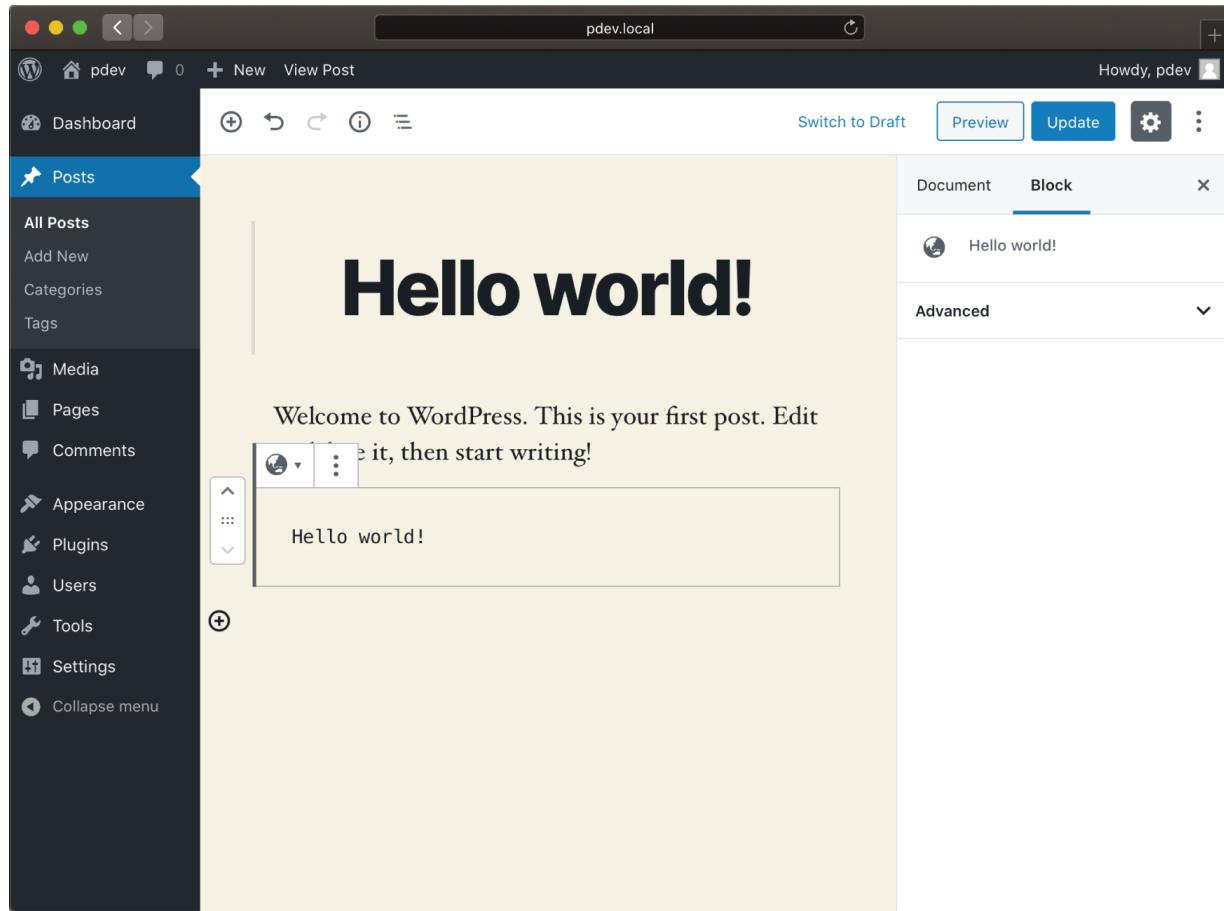


FIGURE 7-14: Selecting our new block

WP-CLI SCAFFOLDING

The WordPress command-line interface, called WP-CLI, provides scaffolding for many different things, and all versions going forward will now support simple scaffolding for blocks.

The one caveat to using this approach currently is that the JavaScript it produces is not ES6, but rather ES5. This is expected to change to ES6 at a later date, but keep this in mind as you use it and explore the code it creates.

You can get a complete list of properties from the [WordPress.org](https://wordpress.org) developer handbook at

<https://developer.wordpress.org/cli/commands/scaffold/block>

Plugin

Create a plugin called pdev2 using the following command:

```
wp scaffold plugin pdev2
```

This creates the `wp-content/plugins(pdev2)` directory and creates a bunch of common default configuration files that are useful for plugins to have.

Blocks

Next let's create two blocks, as shown here:

```
wp scaffold block foo --title="Foo" --plugin=pdev2
wp scaffold block bar --title="Bar" --plugin=pdev2
```

This creates two subdirectories in the plugin directory, one for each block. Inside those directories are an `index.js` file, used to register the block type, and two separate CSS files, one for editor styles and another for theme styles.

Includes

You'll need to manually include the block files in the main plugin file, so open `wp-content/plugins(pdev2(pdev2.php)` and replace this:

```
// Your code starts here.
```

with the following:

```
include __DIR__ . '/blocks/foo.php';
include __DIR__ . '/blocks/bar.php';
```

Activation

Visit your admin area's Plugins page and activate your `pdev2` plugin. Do the same as you did earlier and edit (or add) a post, as shown in [Figure 7-15](#).

[Figure 7-16](#) shows what the block that the WP-CLI scaffolding generated for you looks like in the Gutenberg editor.

Wrap-Up

Wasn't that easy? By using four commands and editing a single file, you got two custom blocks generated for you, without you even needing to worry

about doing any of that complicated tooling stuff (aside from using WP-CLI).

Maybe you're wondering why we spent so much time on tooling simply to have you avoid using it in this section; well, we felt that it was important to communicate how complicated creating a block is. This will give you an immediate appreciation for why many of these tools are actually incredibly useful.

Tooling isn't really part of the WordPress PHP development best practices yet, though it is definitely becoming the status quo as more themes and plugins introduce their own Composer support. If you're like many other professional WordPress plugin developers and you haven't really experienced much in the way of dependency management or code compilation, you'll get a good primer in this book where it makes sense.

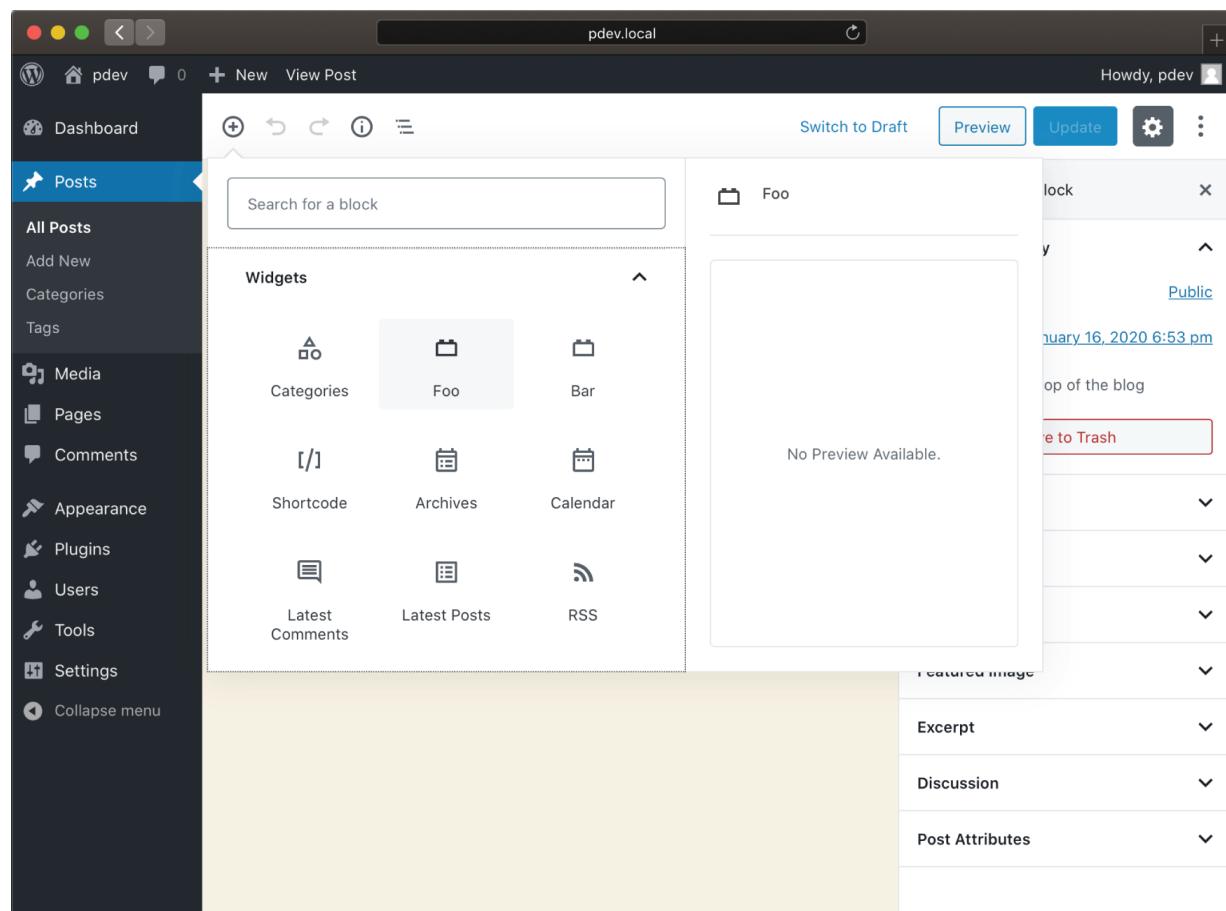


FIGURE 7-15: Editing a post

CREATE-GUTEN-BLOCK TOOLKIT

`create-guten-block` is a zero-configuration toolkit to develop WordPress Gutenberg blocks in a matter of minutes; this means without configuring React, webpack, modern JavaScript (ES6, ES7), ESLint, Babel, and so on.

Because it essentially requires zero configuration, you can always update the underlying project without any changes in your own code. It includes support for React, JSX, ES6, webpack, auto-prefixed CSS, a build script, and hassle-free updates.

`create-guten-block` is the Gutenberg block equivalent of `create-react-app`, aptly named what it is, which suddenly makes a lot of sense, right? It is the clear winner when it comes to taking the majority of the pain away from creating a custom block.

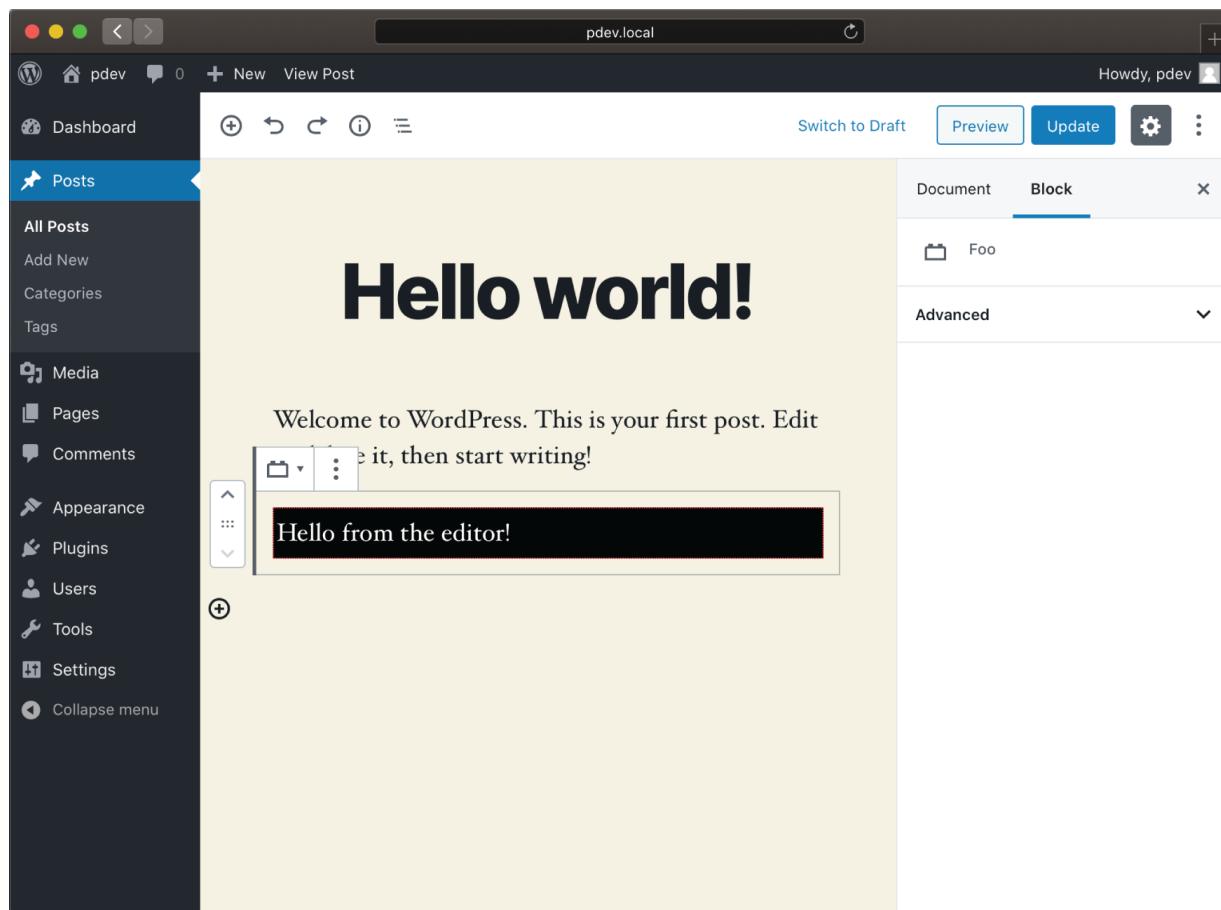


FIGURE 7-16: WP-CLI scaffold generated

Installation

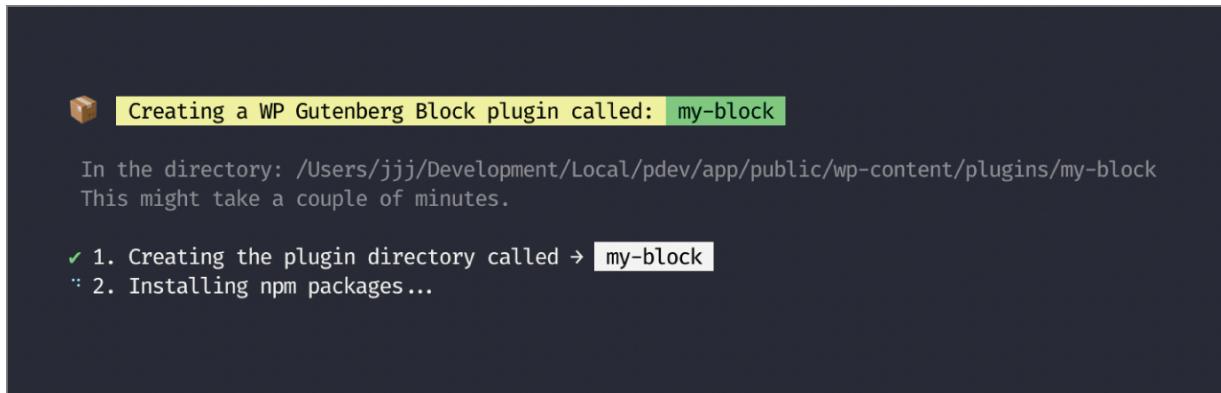
Created and maintained by Ahmad Awais, the `create-guten-block` project has really great documentation, so we're going to send you over there for the instructions. Don't worry, we'll wait here.

<https://github.com/ahmadawais/create-guten-block#getting-started>

You should see the series of screens shown in [Figures 7-19](#).

Activation

As you did previously, activate the `my-block` plugin from within your Plugins page and edit (or add) a post. You will see what is shown in [Figures 7-20](#) and [7-21](#).



```
📦 Creating a WP Gutenberg Block plugin called: my-block
In the directory: /Users/jjj/Development/Local/pdev/app/public/wp-content/plugins/my-block
This might take a couple of minutes.

✓ 1. Creating the plugin directory called → my-block
  2. Installing npm packages...
```

FIGURE 7-17: Build Step 1

Inside that directory, you can run several commands:

👉 Type `npm start`

Use to compile and run the block in development mode.
Watches for any changes and reports back any errors in your code.

👉 Type `npm run build`

Use to build production code for your block inside dist folder.
Runs once and reports back the gzip file sizes of the produced code.

👉 Type `npm run eject`

Removes this tool and copies build dependencies, configuration files and scripts into the plugin folder. ⚡ It's a one way street.
If you do this, you can't go back!

FIGURE 7-18 Build Step 2

Let's build and compile the files...

✓ `Compiled successfully!`

Note that the development build is not optimized.
To create a production build, use `npm run build`

👉 Support Awais via VSCode Power User at <https://VSCode.pro> →

⌚ Watching for changes... (Press CTRL + C to stop).

FIGURE 7-19: Build Step 3

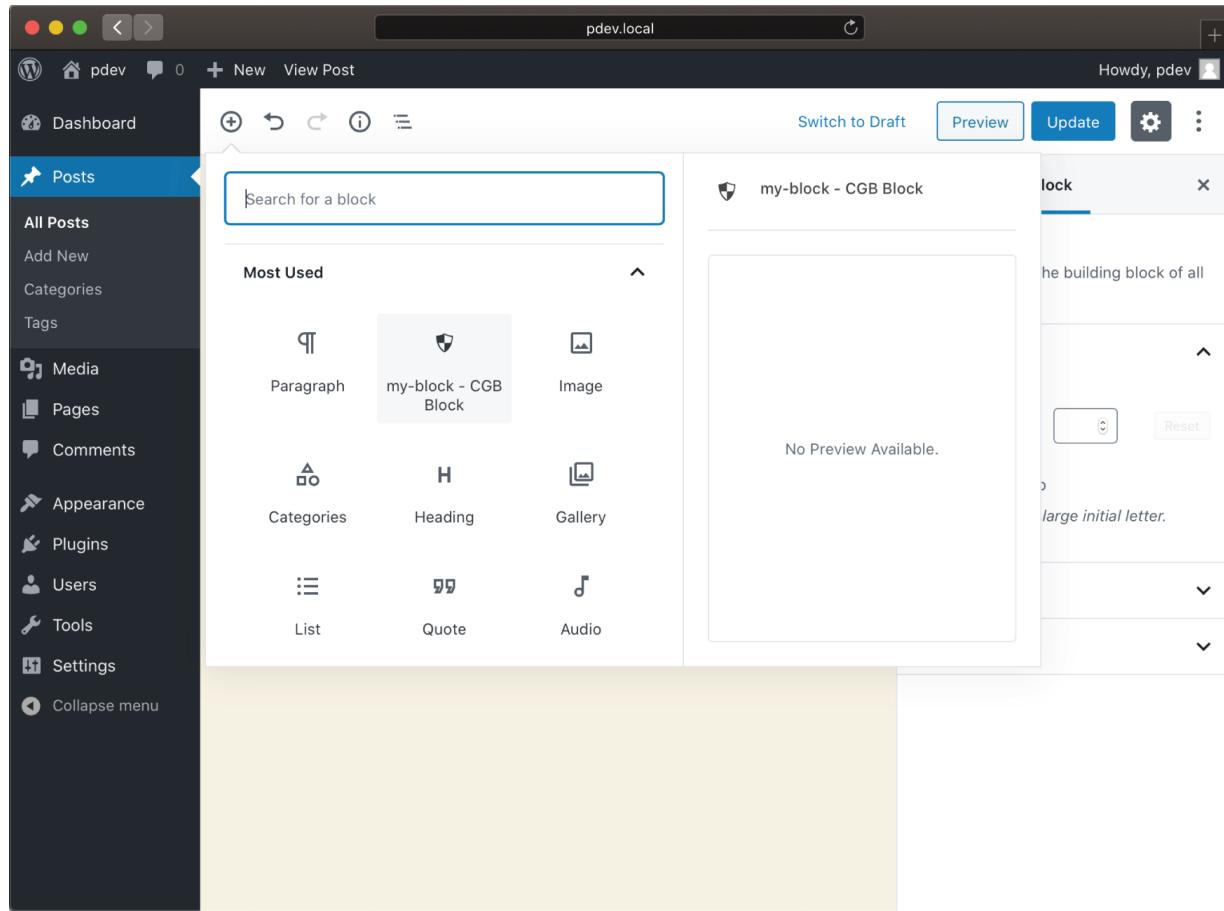


FIGURE 7-20: My Block in Block Library

Wrap-Up

We have a plugin and a block ready to go. We have all of the tooling set up that we could possibly ask for. The sky is the limit now.

BLOCK DIRECTORY

On the roadmap for WordPress 5.5 (by the end of the year 2020) is a Block Directory on WordPress.org much like there is already a Plugin Directory.

The idea is to provide a centralized hub for developers to share their blocks with everyone who is running WordPress, as well as making finding them a breeze. As we're writing this, the only way to discover new blocks is by installing plugins that already have blocks in them, usually by chance or through some kind of referral. Early days!

The Block Directory is still in the design and conception stage, so there aren't any screenshots worth sharing here quite yet. Instead, we're including a few links to the relevant conversations that are happening so you can keep an eye on this going forward:

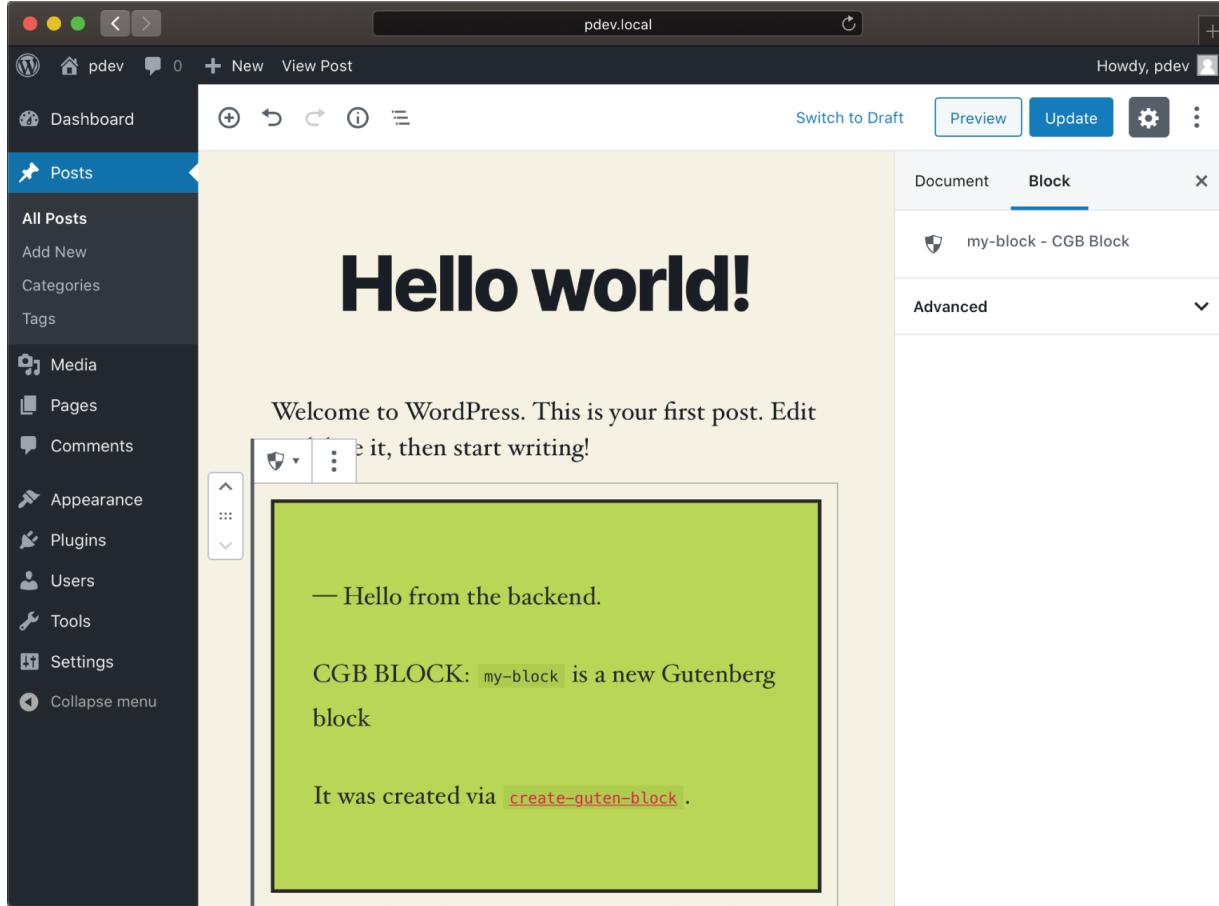


FIGURE 7-21: My Block in Content Area

<https://make.wordpress.org/design/2019/04/02/call-for-design-installing-blocks-from-within-gutenberg>

<https://make.wordpress.org/design/2019/04/26/block-library-installing-blocks-from-within-gutenberg>

<https://make.wordpress.org/design/2019/12/19/block-directory-in-wp-admin-v1>

SUMMARY

Remember, just because Gutenberg is designed to make a user's life easy does not mean it is designed to make a professional plugin developer's life easy. Really, the opposite is true: the less a user needs to do, the more a developer needs to do for them behind the scenes!

JavaScript is an immensely powerful environment. Blocks are the future of WordPress, and blocks require JavaScript, so the future of WordPress is JavaScript, no doubt about it.

8

Content

WHAT'S IN THIS CHAPTER?

- Creating custom post types
- Using custom post types
- Adding and using post metadata
- Building custom post meta boxes
- Using custom taxonomies

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

Posts represent the content of a site in WordPress. In other content management systems, posts are often called *content*. However, WordPress has a legacy of starting as a blogging system, so the terminology stuck. Content is what makes a website a website. It is the most important aspect of the site from the user's perspective because it is the thing that they are building.

Taxonomies in WordPress are a way to classify or categorize posts. Metadata is additional information about a specific post. By combining these two features with posts, you can create any type of website imaginable.

Throughout this chapter, you will work on building a single plugin that houses a user's book collection. The post types, metadata, and taxonomies covered in this chapter contribute to the overall plugin. Nearly every snippet of code provided in the chapter contributes to the plugin by providing a view of how the topics presented work together to manage content.

One important note when dealing with content is that posts, metadata, and taxonomies are typically controlled by themes on the frontend of the site. Depending on what you are building, you may not always have complete control over how the content is output. WordPress theme development is outside the scope of this book, but learning how themes work within the WordPress environment will strengthen your skill set and make you a better developer. In [Chapter 14](#), “The Kitchen Sink,” you will learn more about how plugins can display posts on the frontend.

CREATING CUSTOM POST TYPES

Out of the box, WordPress has several predefined post types that allow users to create and manage the content on their site. These types of content are all the average blogger will ever need.

- **Post:** Blog posts typically presented in order from newest to oldest on the homepage and archives.
- **Page:** Top-level, hierarchical types of content such as About, Contact, and a myriad of other pages.
- **Attachment:** Media attached to other post types, such as image, audio, and video files.
- **Revision:** Revisions of other post types used as a backup system in case a user needs to revert to an older copy.
- **Nav menu item:** Items added to nav menus using WordPress’ built-in menu management system.
- **Reusable block:** One or more blocks from the block editor. When a user saves these reusable blocks, they are stored internally as a post.

As you can see, most of these post types do not match what you might typically think of when thinking of a “post.” Posts can really mean any type of content. For anything beyond a basic site or blog, other post types are necessary for running sites that don’t fit within with this predefined mold. WordPress enables plugin developers to create content types to handle nearly any scenario.

Post Type Possibilities

Custom post types have been available since WordPress 3.0. Over the years, plugin developers have used the system to build a plethora of plugins that do things the creators likely didn't think possible. It shifted WordPress from just being a blogging platform to one of the most robust content management systems in the world. Some plugin developers have launched multimillion-dollar businesses from a starting point of a single custom post type.

You can use custom post types in WordPress to define any type of content. The following is a sample list of ideas, but you should not feel limited to anything but your own imagination:

- Book collection
- Testimonials
- eCommerce products
- Famous quotes
- Event calendar
- Music database
- Image slideshows
- Forums

Registering a Post Type

WordPress makes it easy for plugin developers to create new post types with little code. Before diving in, you need to understand the main function for creating post types and its arguments.

register_post_type

WordPress provides the `register_post_type()` function to create new post types for a site. It is simple to use and gives plugin developers a ton of flexibility by simply configuring its arguments.

```
<?php
register_post_type( string $post_type, array|string $args =
```

```
[] );
```

The function returns a post type object (`WP_Post_Type`) and accepts two parameters.

- `$post_type`: The name of the post type. This should contain only lowercase alphanumeric characters, underscores, and hyphens. The string has a maximum limit of 20 characters.
- `$args`: An array of arguments that configure the post type and how it should be handled within WordPress.

WordPress has more than two dozen arguments for the `$args` parameter, some that have their own subarguments. Each argument provides unique functionality that helps define how your post type will work within the WordPress environment. You can view the entire list via

https://developer.wordpress.org/reference/functions/register_post_type. New options are added over time, so always be sure to reference that list on major updates.

Registering the Book Collection Post Type

Now that you have a basic understanding of how the `register_post_type()` function works, it is time to create your first post type. This is the first step in creating the book collection plugin that you will put together throughout this chapter.

The first step is to create a new folder named `plugin-book-collection` in your test environment's `wp-content/plugins` folder. Then create a `plugin.php` file with the following code in it:

```
<?php
/**
 * Plugin Name: Book Collection
 * Plugin URI: http://example.com/
 * Description: A plugin for managing a book collection.
 * Author: WROX
 * Author URI: http://wrox.com
 */

// Load custom post type functions.
require_once plugin_dir_path( __FILE__ ) . 'post-types.php';
```

This is the primary plugin file. For the purposes of the Book Collection plugin, you will use it to load the various files needed throughout this chapter. The first file is related to registering post types. Now create a `post-types.php` file in your plugin folder.

The first thing you will do with this new file is create a basic custom post type named `book`. The following code snippet has a basic outline of what such a post type might look like. However, you can change the arguments to suit your needs.

```
<?php

add_action( 'init', 'pdev_book_collection_post_types' );

function pdev_book_collection_post_types() {

    register_post_type( 'book', [
        // Post type arguments.
        'public'          => true,
        'publicly_queryable' => true,
        'show_in_rest'      => true,
        'show_in_nav_menus' => true,
        'show_in_admin_bar' => true,
        'exclude_from_search' => false,
        'show_ui'          => true,
        'show_in_menu'      => true,
        'menu_icon'         => 'dashicons-book',
        'hierarchical'     => false,
        'has_archive'       => 'books',
        'query_var'         => 'book',
        'map_meta_cap'      => true,
        // The rewrite handles the URL structure.
        'rewrite' => [
            'slug'      => 'books',
            'with_front' => false,
            'pages'     => true,
            'feeds'      => true,
            'ep_mask'    => EP_PERMALINK,
        ],
        // Features the post type supports.
        'supports' => [
            'title',
            'editor',
        ],
    ] );
}
```

```
        'excerpt',
        'thumbnail'
    ]
);
}
```

The previous code adds a relatively basic post type that works similarly to WordPress posts. It relies on the built-in UI (`show_ui` argument), adds a book collection archive (`has_archive` argument), creates some custom rewrite URL rules (`rewrite` argument), and adds the features that it supports (`supports` argument).

Note that the `pdev_book_collection_post_types()` function is an action that is executed on the `init` action hook. Post types should always be registered on this hook.

Setting Post Type Labels

If you activate the plugin at this point, you will get a new top-level admin menu for managing books. What you might notice is that all of the text labels refer to “posts” instead of “books.” That is not a user-friendly custom post type. You would not want a user to see a View Post link when what you really want them to see is a View Book link, for example.

If you do not set custom labels, WordPress will automatically fall back to the “page” post type labels for hierarchical post types and the “post” post type labels for nonhierarchical types.

To set custom labels, you need to add the `labels` argument to the `$args` array (second parameter) for `register_post_type()`. The `labels` argument should look similar to the following:

```
// Text labels.
'labels' => [
    'name'                      => 'Books',
    'singular_name'              => 'Book',
    'add_new'                    => 'Add New',
    'add_new_item'               => 'Add New Book',
    'edit_item'                  => 'Edit Book',
    'new_item'                   => 'New Book',
    'view_item'                  => 'View Book',
    'view_items'                 => 'View Books',
    'search_items'               => 'Search Books',
    'not_found'                  => 'No books found.',
```

```

        'not_found_in_trash'      => 'No books found in
Trash.',

        'all_items'                => 'All Books',
        'archives'                 => 'Book Archives',
        'attributes'               => 'Book Attributes',
        'insert_into_item'         => 'Insert into book',
        'uploaded_to_this_item'    => 'Uploaded to this

book',

        'featured_image'           => 'Book Image',
        'set_featured_image'       => 'Set book image',
        'remove_featured_image'    => 'Remove book image',
        'use_featured_image'       => 'Use as book image',
        'filter_items_list'        => 'Filter books list',
        'items_list_navigation'    => 'Books list

navigation',

        'items_list'                => 'Books list',
        'item_published'           => 'Book published.',
        'item_published_privately' => 'Book published
privately.',

        'item_reverted_to_draft'   => 'Book reverted to
draft.',

        'item_scheduled'           => 'Book scheduled.',
        'item_updated'              => 'Book updated.

']

```

There are at least 30 possible labels at the time this book was written. However, WordPress expands these labels from time to time as new features are added or things change with the software. You can find the most up-to-date list of labels for the current version of WordPress at https://developer.wordpress.org/reference/functions/get_post_type_labels.

If you activate the plugin at this point, you should see a new Books admin menu and screen, as shown in [Figure 8-1](#).

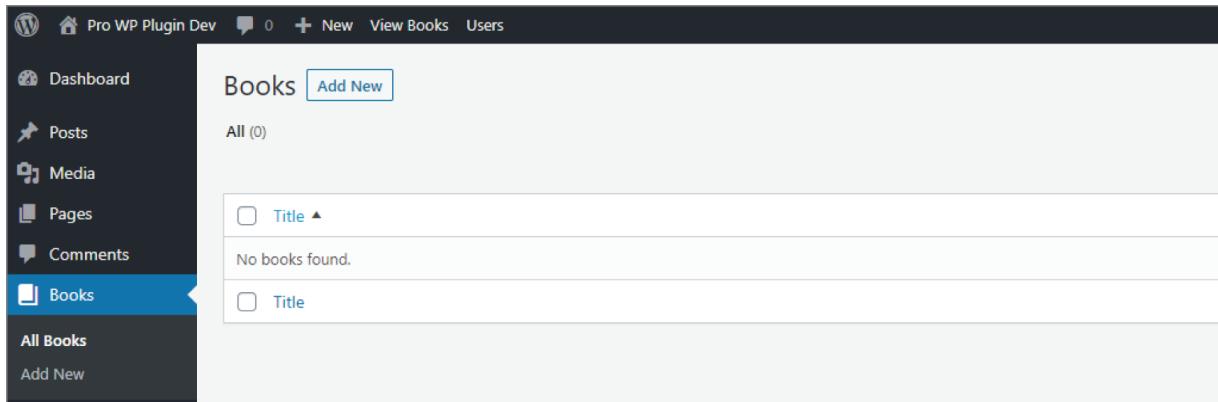


FIGURE 8-1: Books admin menu and screen

Using Custom Capabilities

When building a custom post type, you almost always want to register custom permissions for who can perform what actions. WordPress has a role and capability system for handling permissions, which you will learn more about in [Chapter 9](#), “Users and User Data.” If you add custom capabilities to your post type, the admin screen will not appear for the post type unless your user account has the appropriate capabilities.

For custom post types, it is simply a matter of registering the capabilities as an argument when calling `register_post_type()`. If no custom capabilities are set, WordPress will revert to the “page” post type capabilities for hierarchical post types and “post” capabilities for nonhierarchical types.

The simplest and most effective way to create a suite of capabilities for your book post type is to assign a value to the `capability_type` argument, as shown in the following snippet:

```
'capability_type' => 'book',
```

Generally, you want to use the post type name as the capability name to keep things simple. By adding the preceding code to the arguments array for the post type, WordPress will create the following capabilities:

- `edit_book`
- `read_book`
- `delete_book`

- create_books
- edit_books
- edit_others_books
- edit_private_books
- edit_published_books
- publish_books
- read_private_books
- delete_books
- delete_private_books
- delete_published_books
- delete_others_books

NOTE *If the post type name has an odd plural form (cannot be formed by simply adding an s to the end of the word), such as “boxes” for the “box” post type, you can pass an array to the capability_type argument to handle both the singular and plural forms: ['box', 'boxes']. This will ensure that a capability such as edit_boxes is correctly spelled.*

You may have picked up something different with the first three capabilities in the previous list (edit_book, read_book, delete_book). They are in singular form rather than plural. Generally, a singular form represents what is called a *meta capability* in WordPress. Meta capabilities do something for a specific object, such as a post (book in this case). These capabilities are not assigned to roles and are determined dynamically by mapping them to the other (primitive) capabilities based on the current scenario.

By default, WordPress has automatic mapping disabled. It is rare that you will want this disabled because it will mean having to write a custom mapping filter on the core map_meta_cap filter hook. Instead, you want to enable mapping by setting the map_meta_cap argument to true, as shown in the following code:

```
'map_meta_cap' => true,
```

The preceding code will save you a lot of frustration attempting to manipulate how meta caps work manually.

For most post types, you will only ever need to set the `capability_type` and `map_meta_cap` arguments. However, there are scenarios where you will need more fine-tuned control over the individual capabilities for the post type. In those cases, you will need to set the `capabilities` argument for `register_post_type()`. This will essentially give you manual control over what `capability_type` can do automatically.

The following code snippet is one example of how you could set up your book post type capabilities:

```
'capabilities' => [
    'edit_post'                  => 'edit_book',
    'read_post'                  => 'read_book',
    'delete_post'                => 'delete_book',
    'create_posts'               => 'create_books',
    'edit_posts'                 => 'edit_books',
    'edit_others_posts'          => 'edit_others_books',
    'edit_private_posts'         => 'edit_private_books',
    'edit_published_posts'       => 'edit_published_books',
    'publish_posts'              => 'publish_books',
    'read_private_posts'         => 'read_private_books',
    'read'                        => 'read',
    'delete_posts'               => 'delete_books',
    'delete_private_posts'       => 'delete_private_books',
    'delete_published_posts'     => 'delete_published_books',
    'delete_others_posts'        => 'delete_others_books'
],
```

You don't have to stick to a specific formula with custom post types. Your plugin can mix these up. It can set the same capability for multiple capability options. Or it can set the same capability for each option. For example, you can set each of these capabilities to `manage_books` if you know only certain users will have permission to manage all book collection posts.

Another capability you can set is `do_not_allow` if you don't want to allow access to a specific task. Generally, you wouldn't use this, but some situations may call for it, such as setting `edit_others_posts` to `do_not_allow` so that no user can edit posts created by other users.

WordPress sometimes adds new capabilities to post types. It does not happen often, but it is necessary if something changes with the software. You can view the most updated list of post type capabilities at https://developer.wordpress.org/reference/functions/get_post_type_capabilities.

Attaching Existing Taxonomies

In some cases, you may be building a post type in which you want to use an existing taxonomy. You can easily set this via the `taxonomies` argument within the `$args` array for `register_post_type()`. For example, WordPress has two general taxonomies for blog posts, `category` and `post_tag`, that you may want to use. However, you are not limited to taxonomies created by WordPress. Your post type can use taxonomies created by other plugins too.

Suppose that you wanted to add post tags to book posts. You would set this as shown in the following code:

```
'taxonomies' => [  
    'post_tag'  
,
```

By adding this to your `$args` array for `register_post_type()`, a new Tags submenu item will be added after your Books menu item in the admin, as shown in [Figure 8-2](#).

The screenshot shows the WordPress admin dashboard with a dark theme. The left sidebar has a 'Books' menu item selected, which is highlighted with a blue background. Under 'Books', there are links for 'All Books', 'Add New', 'Tags', 'Appearance', 'Plugins', and 'Users'. The main content area is titled 'Tags' and contains a form for 'Add New Tag'. The form has fields for 'Name' (with a note: 'The name is how it appears on your site.'), 'Slug' (with a note: 'The "slug" is the URL-friendly version of the name. It is usually all lowercase and contains only letters, numbers, and hyphens.'), and 'Description'. To the right of the form is a table listing existing tags. The table has columns for 'Name', 'Description', and checkboxes for 'Bulk Actions'. The tags listed are: 8BIT, alignment, Articles, aside, and audio. Each tag entry includes a note about what it tags posts about.

FIGURE 8-2: Tags submenu item

You will learn how to add custom taxonomies specifically for your post type in the “Creating Custom Taxonomies” section later in this chapter.

POST METADATA

In WordPress, posts can have additional information attached to them. This information is called *post metadata*, or *post meta* for short. It is stored in the `$wpdb->postmeta` table in the database.

Plugins can create, update, and delete meta values in an unlimited number of ways. Traditionally, this is done via a meta box, which you will learn more about in the “Meta Boxes” section of this chapter.

Because you are building a plugin in this chapter for a user to store their book collection, think about the types of metadata related to books. The following are some things you may consider as additional information for the plugin that you are building:

- International Standard Book Number (ISBN)
- Book author
- Goodreads review link
- Amazon affiliate link

- Star rating system

It is important to remember that metadata is not an ideal way to group or categorize posts. The taxonomy system discussed later in this chapter is better for handling features such as categorizing books by genre, for example.

Registering Post Metadata

While it is not strictly a requirement (many plugins skip this step), it is good practice to register any type of metadata your plugin uses with WordPress. To register metadata for a specific post type, you use the `register_post_meta()` function.

```
<?php
register_post_meta( string $post_type, string $meta_key,
array $args );
```

The function accepts the following parameters:

- `$post_type`: The name of the post type
- `$meta_key`: A unique key for storing and retrieving the meta value
- `$args`: An array of optional arguments for further defining the metadata

The following are the possible arguments that can be defined:

- `type`: The type of data allowed for the meta value. `'string'`, `'boolean'`, `'integer'`, and `'number'` are valid.
- `description`: A human-readable description for the meta that may be used.
- `single`: A Boolean value for whether the meta key has one value per object or an array of values.
- `sanitize_callback`: A callback function or method when sanitizing the metadata.
- `auth_callback`: A callback function or method to perform when running meta capability checks to determine whether the user can add,

edit, or delete.

- `show_in_rest`: Whether the metadata is considered public and can be shown via the REST API.

For your book collection plugin, suppose you wanted to keep things simple and allow the user to add the book author's name. You can always register additional metadata later. Create a new file in your plugin folder named `post-meta.php` and make sure to load it from your primary `plugin.php` file. Now add the following code to your new file:

```
<?php

add_action( 'init', 'pdev_books_register_meta' );

function pdev_books_register_meta() {

    register_post_meta( 'book', 'book_author', [
        'single'          => true,
        'show_in_rest'    => true,
        'sanitize_callback' => function( $value ) {
            return wp_strip_all_tags( $value );
        }
    ] );
}
```

The previous code registers a new bit of meta for post objects named `book_author`. It defines a post type of `book`. It also has a single meta value attached to it. However, you may decide to allow for multiple authors for each book and change the `single` argument to `false`.

Adding Post Metadata

WordPress provides the `add_post_meta()` function for adding new metadata to a specific post. The function accepts four parameters.

```
<?php
add_post_meta( int $post_id, string $meta_key, mixed
$meta_value,
bool $unique = false );
```

- `$post_id` : The ID of the post to add metadata to.
- `$meta_key`: The metadata key to add meta value(s) to.

- `$meta_value`: The value attributed to the meta key. Multiple meta values may be added to a single key.
- `$unique`: Whether the meta value provided should be the only meta value. If `true`, there will be only a single meta value. If `false`, multiple meta values can be added. By default, this parameter is set to `false`.

Now that you know how the parameters work for `add_post_meta()`, you can insert some metadata to a specific post. Suppose you have a book post with the ID of `100`. The book is *The Way of Kings*, which was written by Brandon Sanderson. You could attach him as the author with the following code:

```
<?php
add_post_meta( 100, 'book_author', 'Brandon Sanderson', true
);
```

If you were adding multiple authors to a book, such as for *The President Is Missing* by Bill Clinton and James Patterson, you would need to make two calls to `add_post_meta()`. You must also change the `$unique` parameter to `false`. The book post ID in the following example is `200`:

```
<?php
add_post_meta( 200, 'book_author', 'Bill Clinton', false
);
add_post_meta( 200, 'book_author', 'James Patterson', false
);
```

Retrieving Post Metadata

WordPress makes it easy to get post metadata for display or to use in other PHP functions. To retrieve post metadata, use the `get_post_meta()` function, which accepts three parameters.

```
<?php
get_post_meta( int $post_id, string $key = '', bool $single
= false );
```

- `$post_id`: The ID of the post to get the metadata for.
- `$meta_key`: The meta key name to retrieve meta value(s) for.

- `$single`: Whether to return a single meta value (`true`) or return an array of values (`false`). By default, this parameter is set to `false`.

Suppose you wanted to get the book author for *The Way of Kings* from the previous section on adding post metadata. You know the post ID (`100`) and the meta key (`book_author`). You can use the following snippet to return a single author:

```
<?php
$book_author = get_post_meta( 100, 'book_author', true );
// Brandon Sanderson
```

How you handle this for multiple values will be slightly different because `get_post_meta()` will return an array if you set the `$single` parameter to `false`. Now get the authors of *The President Is Missing*, which you added in the previous section, and output the author names as a list.

```
<?php
$book_authors = get_post_meta( 200, 'book_author', false );

echo '<ul>';

foreach ( $book_authors as $author ) {
    printf(
        '<li>%s</li>',
        esc_html( $author )
    );
}

echo '</ul>';
```

The previous code will result in the following HTML output:

```
<ul>
    <li>Bill Clinton</li>
    <li>James Patterson</li>
</ul>
```

Updating Post Metadata

WordPress provides the ability to update post metadata too. You can use the `update_post_meta()` function to update a preexisting meta value, completely overwrite all meta values for a given key, or add new metadata if it does not exist for a post. Many developers use it in lieu of

`add_post_meta()` because it is simpler to use a function that adds or updates metadata.

```
<?php
update_post_meta( int $post_id, string $meta_key, mixed
$meta_value, mixed
$prev_value = '' );
```

- `$post_id`: The post ID to update meta value(s) for.
- `$meta_key`: The meta key to update meta value(s) for.
- `$meta_value`: The new meta value to add to the meta key.
- `$prev_value`: The previous meta value to overwrite. If this parameter is not set, all meta values will be overwritten in favor of the `$meta_value` parameter.

Suppose that the user misspelled Brandon Sanderson's name on their first attempt of adding a book author for *The Way of Kings*. They used an *e* instead of an *a* for his first name. It is simply a matter of adding the new meta value and passing along the old meta value via the `$prev_value` parameter.

```
<?php
update_post_meta( 100, 'book_author', 'Brandon Sanderson',
'Brendon Sanderson' );
```

Because there is only a single book author in this case, you don't actually have to pass the previous value. You can update the book author as shown in the following code snippet:

```
<?php
update_post_meta( 100, 'book_author', 'Brandon Sanderson' );
```

Remember that not setting the `$prev_value` parameter will wipe out all meta values associated with the `book_author` meta key. When you have multiple values for the key, it is often best to update individual values by passing the `$prev_value` into the function. This will ensure that you do not inadvertently delete metadata that you do not intend to delete.

Deleting Post Metadata

There are scenarios in which you will need to delete post metadata completely or to delete a single meta value from a given meta key. WordPress makes this process simple for developers with the `delete_post_meta()` function.

```
<?php
delete_post_meta( int $post_id, string $meta_key, mixed
$meta_value = '' );
```

The function accepts three parameters.

- `$post_id`: The post ID to delete metadata for.
- `$meta_key`: The meta key to delete one or more values for.
- `$meta_value`: The meta value to delete for the given meta key. If this parameter is not set, all meta values for the meta key will be deleted.

In the previous sections, you added, retrieved, and updated the author for *The Way of Kings*, which has a post ID of 100. To delete all authors attached to the book, you would use the following code:

```
<?php
delete_post_meta( 100, 'book_author' );
```

However, the book entry for *The President Is Missing*, which has an ID of 200, has two authors. Using the same method shown previously would delete both authors. If you wanted to delete only a single author from the book, you need to know the exact value and input it as the third parameter. Suppose you wanted to delete Bill Clinton, one of the authors. Use the following code, which would leave the second author attached to the post:

```
<?php
delete_post_meta( 200, 'book_author', 'Bill Clinton' );
```

META BOXES

In the “Post Metadata” section of this chapter, you learned how to handle post metadata via code. However, knowing the appropriate functions will not help your plugin users without an interface. That is where meta boxes

come into play. They provide a way for your plugin users to interact with metadata.

As a plugin developer, it is important to note that the long-term future of meta boxes is a bit uncertain. With the introduction of the block editor, which is covered in [Chapter 7](#), “Blocks and Gutenberg,” developers are encouraged to use the block editor system to handle post metadata. There is no hard deadline for whether or when meta boxes will be deprecated or removed altogether from WordPress. The APIs related to the editor are also constantly being improved. It is important to keep this in mind when deciding how to present metadata options to users.

What Is a Meta Box?

Meta boxes are “boxes” within the post editing screen that allow users to edit metadata. Technically, anything can exist within a meta box. You can use PHP or JavaScript to output any arbitrary HTML output, but the most common use case is building form fields for users to edit custom metadata from plugins. In essence, meta boxes are a UI element.

Adding a Custom Meta Box

To add a custom meta box to the post editing screen, you use the `add_meta_box()` function.

```
<?php
add_meta_box(
    string $id,
    string $title,
    callable $callback,
    string|array|WP_Screen $screen = null,
    string $context = 'advanced',
    string $priority = 'default',
    array $callback_args = null
);
```

The function accepts the following parameters.

- `$id`: A unique ID for the meta box.
- `$title`: A text label shown at the top of the meta box.

- `$callback`: A function or class method that will output the meta box content.
- `$screen`: The screen or screens to output the meta box on. For post meta boxes, this is the name of the post type.
- `$context`: Where to output the meta box by default on the editor screen. 'advanced', 'normal', and 'side' are valid values.
- `$priority`: The priority of the meta box in comparison to other boxes. Valid values are 'default', 'high', and 'low'.
- `$callback_args`: An optional array of custom data to pass as the second parameter to your callback function or method.

Meta boxes need to be registered on a specific hook. Generally, you could register them on the `add_meta_boxes` hook. However, for custom post types, it is best to register on the `add_meta_boxes_{$post_type}` hook, where `$post_type` is the name of the specific post type.

For the plugin that you are building along in this chapter, you need to create a new file named `meta-boxes.php` in your plugin folder and load it, as shown in the following snippet from your main `plugin.php` file:

```
require_once plugin_dir_path( __FILE__ ) . 'meta-boxes.php';
```

In your new `meta-boxes.php` file, use the following code to register a new meta box on the `add_meta_boxes_book` hook:

```
<?php

add_action( 'add_meta_boxes_book',
'pdev_book_register_meta_boxes' );

function pdev_book_register_meta_boxes() {

    add_meta_box(
        'pdev-book-details',
        'Book Details',
        'pdev_book_details_meta_box',
        'book',
        'advanced',
        'high'
    );
}
```

As you can see, it is just a simple call to `add_meta_box()` within the function attached to the action hook. It registers a new meta box with an ID of `pdev-book-details`. Most of the parameters can be configured to your preference. The most important parameter from the function is the callback function, `pdev_book_details_meta_box`, which handles the output of the meta box content.

A meta box can contain any number of form fields, but for this meta box, let's stick with outputting a field for the user to enter a single book author. In your plugin's `meta-boxes.php` file, add the following callback function for the meta box:

```
function pdev_book_details_meta_box( $post ) {

    // Get the existing book author.
    $author = get_post_meta( $post->ID, 'book_author',
true );

    // Add a nonce field to check on save.
    wp_nonce_field( basename( __FILE__ ), 'pdev-book-
details' ); ?>

    <p>
        <label>
            Book Author:
            <br/>
            <input type="text" name="pdev-book-
author"
                value=<?php echo esc_attr( $author ); ?>
        </label>
    </p>

<?php }
```

One thing you should note about the previous code is the `$post` parameter for the `pdev_book_details_meta_box()` function. This parameter is automatically passed into the callback function for meta boxes and is the `WP_Post` object for the post.

The first thing the preceding code does is get the current book author value attached to the book and assign it to the `$author` variable. If this is a new post or an author has not yet been set, the value will be an empty string.

The second bit of code is a call to `wp_nonce_field()`, which outputs a hidden HTML form field with a nonce value. You will later check this value for security purposes when saving metadata to the database. Nonces are covered in more detail in [Chapter 4](#), “Security and Performance.”

The final thing the function does is output a text input field to allow the plugin user to enter a book author's name. The `value` attribute is an escaped copy of the `$author` variable.

Saving Meta Box Data

Creating a custom meta box is one part of a two-part process of handling the user experience aspect of manipulating post metadata. The second part is adding, updating, and/or deleting metadata when a post is saved. This is the point where everything you learned in the “Post Metadata” section of this chapter becomes useful in practice.

To save metadata from your custom meta box fields, you need to find the appropriate hook. There are several hooks that fire during the post-saving procedure, and many theme authors rely directly on the `save_post` hook. However, it is not always the best option because it runs for every post type. Instead, when you are saving metadata for only a specific post type, use the `save_post_{$post_type}` hook where `$post_type` is the name of the post type. Aside from their different names and when they are fired, the two hooks are identical.

```
<?php
do_action( "save_post_{$post_type}", int $post_id, WP_Post
$post, bool $update );
```

The hook passes up to three parameters:

- `$post_id`: The ID of the post currently being saved
- `$post`: The post object for the current post
- `$update`: A Boolean that determines if the current post is being updated (`true`) or is a new post being saved for the first time (`false`)

For saving the book author, you need to add a custom action to the `save_post_book` action hook and require the first two parameters, `$post_id`

and \$post, which provided necessary data.

Append the following code snippet to your `meta-boxes.php` file in your book plugin folder:

```
add_action( 'save_post_book', 'pdev_book_save_post', 10, 2
);

function pdev_book_save_post( $post_id, $post ) {

    // Verify the nonce before proceeding.
    if (
        ! isset( $_POST['pdev-book-details'] ) ||
        ! wp_verify_nonce( $_POST['pdev-book-details'],
        basename( __FILE__ ) )
    ) {
        return;
    }

    // Bail if user doesn't have permission to edit the
    post.
    if ( ! current_user_can( 'edit_post', $post_id ) ) {
        return;
    }

    // Bail if this is an Ajax request, autosave, or
    revision.
    if (
        wp_is_doing_ajax() ||
        wp_is_post_autosave( $post_id ) ||
        wp_is_post_revision( $post_id )
    ) {
        return;
    }

    // Get the existing book author if the value exists.
    // If no existing book author, value is empty string.
    $old_author = get_post_meta( $post_id, 'book_author',
    true );

    // Strip all tags from posted book author.
    // If no value is passed from the form, set to empty
    string.
    $new_author = isset( $_POST['pdev-book-author'] ) ?
        wp_strip_all_tags( $_POST['pdev-book-
    author'] ) :
        '';
}
```

```

// If there's an old value but not a new value,
delete old value.
if ( ! $new_author && $old_author ) {
    delete_post_meta( $post_id, 'book_author' );

    // If the new value doesn't match the new value,
    // add/update.
} elseif ( $new_value !== $old_value ) {
    update_post_meta( $post_id, 'book_author',
    $new_value );
}
}

```

The preceding code can essentially be broken into three sections. The first section of the code is running several checks to determine whether to proceed. Once the code runs through its checks, the second section gets the data it needs. Finally, it can move on to saving or deleting metadata.

The most important check comes first, which verifies that the nonce field created in the previous “Adding a Custom Meta Box” section is valid. The second check determines whether the current user has permission to edit the post at all. The third check is kind of a stock check that you should add when saving metadata. Posts are saved in all sorts of situations in WordPress, even when not on the edit post screen. Therefore, you must determine that the post is not being saved as part of an Ajax request, auto-save is not running, and the post is not a revision.

After all the checks have passed, the previous code grabs the existing book author metadata via the `get_post_meta()` function. If no value exists in the database, the function will return an empty string. Then, it checks `$_POST['pdev-book-author']` to get the new book author posted from the HTML text input box. That data must be run through a sanitizing function to make sure it is safe. In this case, `wp_strip_all_tags()` works because there is no need for HTML.

After getting both the old and new book author metadata, it is a simple matter of running a conditional check to determine what to do with the data. If the new meta value is empty and an old value exists, use `delete_post_meta()` to delete it. Else, if the new value does not equal the old value, use `update_post_meta()` to update it. Also, remember that `update_post_meta()` will add a new meta value if it does not yet exist.

CREATING CUSTOM TAXONOMIES

Taxonomies are a way to group or categorize objects in WordPress. Technically, they can group any type of object, such as posts, comments, or users. However, the majority of plugins use taxonomies to group posts together.

WordPress ships with several taxonomies by default.

- **Category:** A hierarchical taxonomy used to categorize blog posts
- **Post Tag:** A nonhierarchical taxonomy used to tag blog posts
- **Link Category:** A nonhierarchical taxonomy used to categorize links
- **Nav Menu:** A nonhierarchical taxonomy that represents navigation menus and groups nav menu items

The true power of custom taxonomies is creating them to use alongside custom post types. Often, it is necessary to create custom taxonomies with post types so that users have a way to organize individual posts of the post type.

Understanding Taxonomies

To understand how taxonomies work, you must first understand that an individual taxonomy is a group of terms. Each term within the taxonomy is what technically groups posts together.

In this chapter, you are creating a new custom post type: book. A user would use the book post type to organize their personal collection of books. Your plugin user will likely want some way to organize their book collection into groups. This is where custom taxonomies fit into the picture. Some possible taxonomies for books follow:

- Genre (fantasy, romance, science fiction)
- Format (hardcover, paperback, digital)
- Publisher (Wiley, Tor Books)
- Form (novel, novella)

Each of these taxonomies would enable users to label their books with information that further defines the content. Essentially, taxonomies provide clearer organization and definition for the content.

This section focuses on creating the genre taxonomy for the book post type, which you created earlier in this chapter.

NOTE *This chapter focuses solely on creating taxonomies for post types because this will be the scenario they'll be used for in most cases. However, you can add taxonomies to any object type in WordPress, such as comments and users.*

Registering a Custom Taxonomy

WordPress makes it easy for plugin developers to register a custom taxonomy with a single function. Like registering a custom post type, it is simply a matter of calling the function and adding custom parameters.

register_taxonomy

WordPress provides the `register_taxonomy()` function for registering a new taxonomy with the system. This function enables you to create a new taxonomy and set it up by using custom arguments that define how the taxonomy should be handled within WordPress.

```
<?php
register_taxonomy( string $taxonomy, array|string
$object_type, array|string
$args = [] );
```

The function accepts three parameters.

- `$taxonomy`: The name of your plugin's taxonomy. This should contain only alphanumeric characters and underscores.
- `$object_type`: A single object or an array of objects to add the taxonomy to. For post objects, this is the post type name.
- `$args`: An array of arguments that defines how WordPress should handle your taxonomy.

There are many flags that you can set for the \$args parameter, which is more than can be covered in this chapter. You can view all arguments via the developer documentation:

https://developer.wordpress.org/reference/functions/register_taxonomy.

Registering the Genre Taxonomy

Now that you've reviewed the `register_taxonomy()` parameters and arguments, it is time to use that knowledge to create new taxonomies.

First, you need to create a new `taxonomies.php` file within your plugin folder. Then, append the following line of code to your primary `plugin.php` file to load it:

```
require_once plugin_dir_path( __FILE__ ) . 'taxonomies.php';
```

The next code snippet will register a hierarchical taxonomy named `genre`, which users can use to organize their book collection. Add the following code to your `taxonomies.php` plugin file:

```
<?php

add_action( 'init', 'pdev_books_register_taxonomies' );

function pdev_books_register_taxonomies() {

    register_taxonomy( 'genre', 'book', [
        // Taxonomy arguments.
        'public'          => true,
        'show_in_rest'    => true,
        'show_ui'         => true,
        'show_in_nav_menus' => true,
        'show_tagcloud'   => true,
        'show_admin_column' => true,
        'hierarchical'    => true,
        'query_var'       => 'genre',
        // The rewrite handles the URL structure.
        'rewrite' => [
            'slug'          => 'genre',
            'with_front'    => false,
            'hierarchical' => false,
            'ep_mask'       => EP_NONE
        ]
    ]
}
```

```
        ],
        // Text labels.
        'labels'          => [
            'name'           =>
            'Genres',        => 'Genre',
            'Genres',        => 'Genre',
            'Genres',        => 'Genre',
            'Genres',        => 'Search',
            'Genres',        => 'Popular',
            'Genres',        => 'All',
            'Genres',        => 'Edit',
            'Genre',          => 'View',
            'Genre',          => 'Update',
            'Genre',          => 'Add New',
            'Genre',          => 'New',
            'Genre Name',    => 'No',
            'genres found.',=> 'No',
            'genres',         => 'Genres',
            'list navigation',=> 'Genres',
            'list',           => 'Genres',
            // Hierarchical only.
            'select_name'    => 'Select',
            'parent_item'    => 'Parent',
            'parent_item_colon'=> 'Parent',
            'Genre:'          =>
        ]
    );
}
```

After you add the preceding code, you'll be presented with a new submenu item under the Books menu item in the admin, labeled Genres. You also have a new meta box for assigning genres to individual books, as shown in [Figure 8-3](#). WordPress will automatically generate this meta box.

WARNING *Your function for registering new taxonomies must be added to the `init` action hook for the taxonomies to be properly registered.*

Assigning a Taxonomy to a Post Type

Rather than registering a custom taxonomy, sometimes you may need to assign a taxonomy to a post type. If your plugin creates the taxonomy, you would do this with the `register_taxonomy()` function. If your plugin is creating a custom post type but needs to add an existing taxonomy, you would use the `taxonomy` argument for `register_post_type()`. However, there are scenarios where you either need or can choose to use an alternative solution.

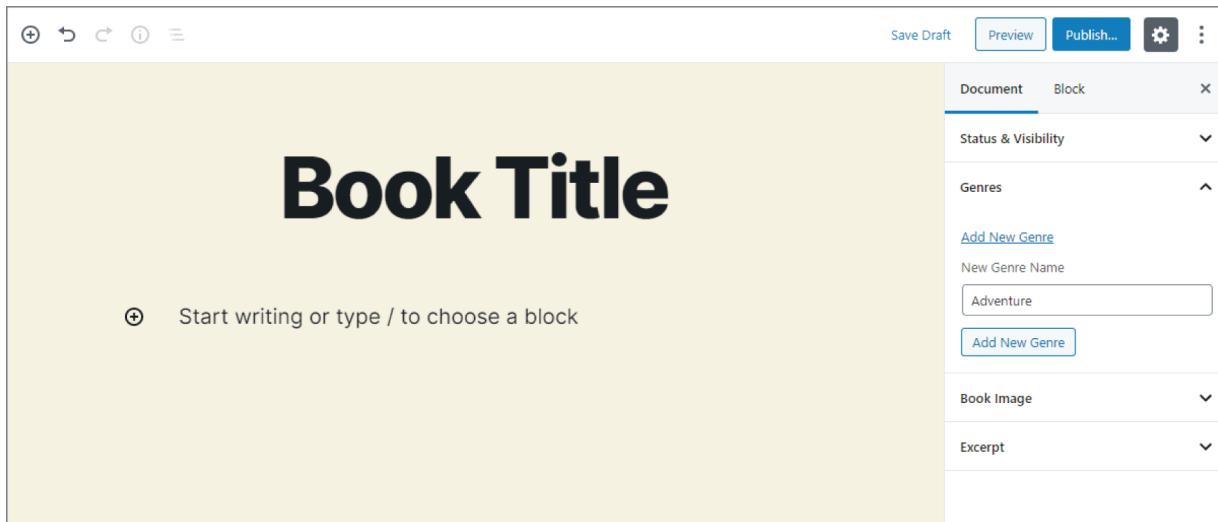


FIGURE 8-3: Genres submenu

WordPress provides the `register_taxonomy_for_object_type()` function, which enables you to set a taxonomy on any object type, which will typically be a specific post type.

```
<?php
register_taxonomy_for_object_type( string $taxonomy, string
$object_type );
```

The function accepts two parameters.

- `$taxonomy`: The name of the taxonomy your plugin will add to the object type.
- `$object_type`: The name of the object type to add the taxonomy to. Most of the time, this will be a post type.

Suppose you are working for a client who has an existing plugin that already registers the genre taxonomy you added in the “Registering a Custom Taxonomy” section of this chapter. Instead of registering the taxonomy in your plugin, you can simply add the taxonomy to your new book post type, as shown in the next code snippet:

```
<?php
add_action( 'init', 'pdev_add_genres_to_books' );

function pdev_add_genres_to_books() {
    register_taxonomy_for_object_type( 'genre', 'book' );
}
```

USING CUSTOM TAXONOMIES

Like with custom post types, taxonomies are most often output via WordPress theme template files. However, there are scenarios where your plugin may need to use taxonomy functions for displaying information.

Retrieving a Taxonomy

When you register a custom taxonomy with WordPress, it generates a new `WP_Taxonomy` object and stores the value globally with all the other taxonomy objects. In some cases, you may need to retrieve information about a registered taxonomy. The object's public properties are the arguments supplied for the `$args` array when you first registered the taxonomy with `register_taxonomy()`.

The `get_taxonomy()` function allows you to get a specific `WP_Taxonomy` object. The function accepts a single parameter of `$taxonomy`, which should be the name of the taxonomy.

```
<?php
get_taxonomy( string $taxonomy );
```

Suppose you need to display the singular name label for the genre taxonomy that you registered. You would use the following code to get the taxonomy object and display this label:

```
<?php
// Get the genre taxonomy object.
$genre = get_taxonomy( 'genre' );

// Prints "Genre".
echo $genre->labels->singular_name;
```

Using a Taxonomy with Posts

When using taxonomy with posts, you will generally be listing the taxonomy terms for the given post alongside some or all of the content of the post. This would allow viewers to note there is a taxonomy for the post and allow them to find related posts by a given taxonomy term.

WordPress provides the `the_terms()` function for displaying a specific post's terms.

```
<?php
the_terms(
    int $post_id,
    string $taxonomy,
    string $before = '',
    string $sep = ', ',
    string $after = ''
);
```

The function accepts five parameters.

- `$id`: The ID of the post to list the taxonomy's terms for.
- `$taxonomy`: The name of the taxonomy to list terms for.
- `$before`: Content to display before the list of terms.

- `$sep`: Any string of text or HTML to separate individual terms in the list. This defaults to a comma followed by a space.
- `$after`: Content to display after the list of terms.

The `the_terms()` function is a wrapper function for `get_the_term_list()`. The former function displays the list of terms for the taxonomy, and the latter returns them for use in PHP. Their parameters are identical.

Assuming you want to display the genres of the book post currently being viewed, you would use the following:

```
<?php
the_terms(
    get_the_ID(),
    'genre',
    '<div class="pdev-book-genres">',
    '',
    '</div>'
);
```

If you needed to return the value instead of immediately printing it to the screen, use the next code snippet:

```
<?php
$genres = get_the_term_list(
    get_the_ID(),
    'genre',
    '<div class="pdev-book-genres">',
    '',
    '</div>'
);
```

Taxonomy Conditional Tags

WordPress has a few conditional tags for taxonomies. Conditional tags check a specific condition and return `true` if the condition is met or `false` if the condition is not met.

taxonomy_exists

The `taxonomy_exists()` function checks whether a taxonomy has been registered with WordPress. It accepts a single parameter of `$taxonomy`, which should be the name of the taxonomy you're checking.

```
<?php
taxonomy_exists( string $taxonomy );
```

Suppose you wanted to check whether the genre taxonomy exists before attempting to register a custom taxonomy of your own. You can use the following code to determine whether to register a custom taxonomy or register an existing taxonomy to your book post type:

```
<?php
add_action( 'init', 'pdev_maybe_register_genre' );

function pdev_maybe_register_genre() {

    if ( taxonomy_exists( 'genre' ) ) {
        register_taxonomy_for_object_type( 'genre',
        'book' );
    } else {
        register_taxonomy( 'genre', 'book' );
    }
}
```

is_taxonomy_hierarchical

The `is_taxonomy_hierarchical()` function determines whether a given taxonomy is hierarchical. It accepts a single parameter of `$taxonomy`, which should be the name of the taxonomy.

```
<?php
is_taxonomy_hierarchical( string $taxonomy );
```

Imagine that you wanted to display a list of terms using the `wp_list_categories()` function if a taxonomy was hierarchical but a tag cloud using the `wp_tag_cloud()` function if the taxonomy was flat. You could run an `if/ else` check with the result of the `is_taxonomy_hierarchical()` function as the condition to check.

```
<?php
$taxonomy = 'genre';

// If taxonomy is hierarchical, print list.
if ( is_taxonomy_hierarchical( $taxonomy ) ) {
    printf(
        '<ul>%s</ul>',
        wp_list_categories( [
            'taxonomy' => $taxonomy,
```

```

        'title_li' => '',
        'echo'      => false
    ] )
);

// If taxonomy is flat, print tag cloud.
} else {
    printf(
        '%s',
        wp_tag_cloud( [
            'taxonomy' => $taxonomy,
            'echo'      => false
        ] )
    );
}

```

is_tax

The `is_tax()` function determines whether a site visitor is on a term archive page on the frontend of the site. The function would more appropriately be named something like `is_taxonomy_term_archive()`, which is what it actually checks. However, `is_tax()` is a legacy function name that WordPress has stuck with.

When using no parameters, the function simply checks whether the visitor is on any taxonomy term archive. However, you may optionally set one or both of its parameters for a more specific check.

```
<?php
is_tax( string|array $taxonomy = '', int|string|array $term
= '' );
```

The following two parameters are accepted:

- `$taxonomy`: The name of the taxonomy to check for
- `$term`: The name of the term from the taxonomy to check for

With the next snippet of code, you display one of three messages depending on the condition that is true. You first check to see whether the visitor is on the fantasy genre archive page. If that is not the case, you check to see whether the visitor is on any genre archive page. Finally, if the first two checks fail, you check whether the visitor is on any taxonomy term archive page.

```

<?php
// If viewing a specific genre archive.
if ( is_tax( 'genre', 'fantasy' ) ) {
    echo 'You are viewing the fantasy genre archive';

// If viewing any genre term archive.
} elseif ( is_tax( 'genre' ) ) {
    echo 'You are viewing a genre archive';

// If viewing any taxonomy term archive.
} elseif ( is_tax() ) {
    echo 'You are viewing a taxonomy term archive.';
}

```

A POST TYPE, POST METADATA, AND TAXONOMY PLUGIN

Throughout this chapter, you have been building the pieces of a book collection plugin. This plugin allows users to input the books they own, add the book author's name, and organize the books by genre. It is a simple plugin that you can further expand upon to make a more advanced book collection plugin. You can also use it as a template to create other custom post types, metadata, and taxonomies.

Review what the full plugin code should look like when it is finished. Each file of the plugin should be within the `plugin-book-collection` folder.

The primary `plugin.php` file, which loads the other plugin files, should look like the following:

```

<?php
/**
 * Plugin Name: Book Collection
 * Plugin URI:  http://example.com/
 * Description: A plugin for managing a book collection.
 * Author:      WROX
 * Author URI:  http://wrox.com
 */

// Load custom post type functions.
require_once plugin_dir_path( __FILE__ ) . 'post-types.php';
require_once plugin_dir_path( __FILE__ ) . 'post-meta.php';
require_once plugin_dir_path( __FILE__ ) . 'meta-boxes.php';
require_once plugin_dir_path( __FILE__ ) . 'taxonomies.php';

```

The `post-types.php` file, which registers the book post type, should look like the next snippet:

```
<?php

add_action( 'init', 'pdev_book_collection_post_types' );

function pdev_book_collection_post_types() {

    register_post_type( 'book', [
        // Post type arguments.
        'public'              => true,
        'publicly_queryable'  => true,
        'show_in_rest'         => true,
        'show_in_nav_menus'   => true,
        'show_in_admin_bar'   => true,
        'exclude_from_search' => false,
        'show_ui'              => true,
        'show_in_menu'         => true,
        'menu_icon'            => 'dashicons-book',
        'hierarchical'         => false,
        'has_archive'          => 'books',
        'query_var'            => 'book',
        'map_meta_cap'         => true,
        // The rewrite handles the URL structure.
        'rewrite' => [
            'slug'      => 'books',
            'with_front' => false,
            'pages'     => true,
            'feeds'     => true,
            'ep_mask'   => EP_PERMALINK,
        ],
        // Features the post type supports.
        'supports' => [
            'title',
            'editor',
            'excerpt',
            'thumbnail'
        ],
        // Text labels.
        'labels'          => [
            'name'           => 'Books',
            'singular_name' => 'Book',
        ]
    ] );
}
```

'add_new' => 'Add New',
'add_new_item' => 'Add New'
Book', 'edit_item' => 'Edit
Book', 'new_item' => 'New
Book', 'view_item' => 'View
Book', 'view_items' => 'View
Books', 'search_items' => 'Search
Books', 'not_found' => 'No books
found.', 'not_found_in_trash' => 'No books
found in Trash.', 'all_items' => 'All
Books', 'archives' => 'Book
Archives', 'attributes' => 'Book
Attributes', 'insert_into_item' => 'Insert
into book', 'uploaded_to_this_item' => 'Uploaded
to this book', 'featured_image' => 'Book
Image', 'set_featured_image' => 'Set book
image', 'remove_featured_image' => 'Remove
book image', 'use_featured_image' => 'Use as
book image', 'filter_items_list' => 'Filter
books list', 'items_list_navigation' => 'Books
list navigation', 'items_list' => 'Books
list', 'item_published' => 'Book
published.', 'item_published_privately' => 'Book
published privately.', 'item_reverted_to_draft' => 'Book
reverted to draft.', 'item_scheduled' => 'Book'

```

        scheduled.',           'item_updated'          => 'Book
updated.'
    ]
]
);
}

```

Next, check the `post-meta.php` file, which registers the custom post metadata.

```

<?php

add_action( 'init', 'pdev_books_register_meta' );

function pdev_books_register_meta() {

    register_post_meta( 'book', 'book_author', [
        'single'          => true,
        'show_in_rest'    => true,
        'sanitize_callback' => function( $value ) {
            return wp_strip_all_tags( $value );
        }
    ]);
}

```

Then, review your meta box code in the `meta-boxes.php` file, which creates the UI for the user to enter and save their book metadata.

```

<?php

add_action( 'add_meta_boxes_book',
'pdev_book_register_meta_boxes' );

function pdev_book_register_meta_boxes() {

    add_meta_box(
        'pdev-book-details',
        'Book Details',
        'pdev_book_details_meta_box',
        'book',
        'advanced',
        'high'
    );
}

function pdev_book_details_meta_box( $post ) {

```

```
        // Get the existing book author.
        $author = get_post_meta( $post->ID, 'book_author',
true );

        // Add a nonce field to check on save.
        wp_nonce_field( basename( __FILE__ ), 'pdev-book-
details' ); ?>

<p>
    <label>
        Book Author:
        <br/>
        <input type="text" name="pdev-book-
author"
            value=<?php echo esc_attr( $author ); ?>
        </label>
    </p>

<?php }

add_action( 'save_post_book', 'pdev_book_save_post', 10, 2
);

function pdev_book_save_post( $post_id, $post ) {

    // Verify the nonce before proceeding.
    if (
        ! isset( $_POST['pdev-book-details'] ) ||
        ! wp_verify_nonce( $_POST['pdev-book-details'],
basename( __FILE__ ) )
    ) {
        return;
    }

    // Bail if user doesn't have permission to edit the
post.
    if ( ! current_user_can( 'edit_post', $post_id ) ) {
        return;
    }

    // Bail if this is an Ajax request, autosave, or
revision.
    if (
        wp_is_doing_ajax() ||
        wp_is_post_autosave( $post_id ) ||
        wp_is_post_revision( $post_id )
    ) {
```

```

        return;
    }

    // Get the existing book author if the value exists.
    // If no existing book author, value is empty string.
    $old_author = get_post_meta( $post_id, 'book_author',
true );

    // Strip all tags from posted book author.
    // If no value is passed from the form, set to empty
string.
    $new_author = isset( $_POST['pdev-book-author'] )
                  ? wp_strip_all_tags( $_POST['pdev-book-
author'] )
                  : '';

    // If there's an old value but not a new value,
delete old value.
    if ( ! $new_author & $old_author ) {
        delete_post_meta( $post_id, 'book_author' );

        // If the new value doesn't match the new value,
add/update.
        } elseif ( $new_value !== $old_value ) {
            update_post_meta( $post_id, 'book_author',
$new_value );
        }
    }
}

```

Finally, review the `taxonomies.php` file, which is used for registering the custom genre taxonomy for organizing books in the collection.

```

<?php

add_action( 'init', 'pdev_books_register_taxonomies' );

function pdev_books_register_taxonomies() {

    register_taxonomy( 'genre', 'book', [
        // Taxonomy arguments.
        'public'          => true,
        'show_in_rest'    => true,
        'show_ui'         => true,
        'show_in_nav_menus' => true,
        'show_tagcloud'   => true,
        'show_admin_column' => true,

```

```

'hierarchical'      => true,
'query_var'        => 'genre',

// The rewrite handles the URL structure.
'rewrite' => [
    'slug'          => 'genre',
    'with_front'   => false,
    'hierarchical' => false,
    'ep_mask'       => EP_NONE
],
// Text labels.
'labels'          => [
    'name'          =>
'Genres',
    'singular_name' => 'Genre',
    'menu_name'      =>
'Genres',
    'name_admin_bar' => 'Genre',
    'search_items'   => 'Search
Genres',
    'popular_items'  => 'Popular
Genres',
    'all_items'      => 'All
Genres',
    'edit_item'       => 'Edit
Genre',
    'view_item'       => 'View
Genre',
    'update_item'     => 'Update
Genre',
    'add_new_item'    => 'Add New
Genre',
    'new_item_name'   => 'New
Genre Name',
    'not_found'       => 'No
genres found.',
    'no_terms'        => 'No
genres',
    'items_list_navigation' => 'Genres
list navigation',
    'items_list'       => 'Genres
list',
    // Hierarchical only.
    'select_name'     => 'Select
Genre',
    'parent_item'      => 'Parent

```

```
Genre',
        'parent_item_colon'          => 'Parent
Genre:''
    ]
];
}
```

Now you have built a fully functional plugin that makes use of custom post types, post metadata, and taxonomies. This is just the beginning, though. There is an entire world of possibilities when it comes to building custom content solutions for your plugins. Use this plugin as a basic template to explore building custom content of your own.

SUMMARY

This chapter represents a small sampling of what's possible with custom post types, post metadata, and taxonomies. The biggest lesson you should take away is that you can use WordPress to create and manage literally any type of content you can imagine. The platform isn't simply limited to blog posts and pages.

The information presented in this chapter reveals many possibilities for plugin developers. You can make plugins for public use to give thousands of people new ways to manage content, or you can use these tools for custom client websites that have unique content needs.

9

Users and User Data

WHAT'S IN THIS CHAPTER?

- Working with users and user functions
- Adding, updating, and retrieving user data
- Understanding roles and capabilities
- Limiting access with user permissions
- Building custom roles and capabilities

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

When WordPress was first launched more than 15 years ago, it was primarily a blogging system used by individuals. Over the years, it has become more of a content management system (CMS) that powers nearly any type of site conceivable. A large part of what is possible now centers on its user system. Today, WordPress powers websites with thousands, hundreds of thousands, and even millions of users.

Knowing how WordPress handles users is now an important tool in any plugin developer's toolbox. You will deal with various user scenarios in many of your plugins. Perhaps, more important, understanding the roles and capabilities system is paramount to developing a solid and secure plugin. Roles and capabilities define what users can do within individual sites. They represent the permissions system for WordPress.

People use WordPress for private membership sites, social networks, online journalism, medical databases, education centers, and much more. Each of these types of websites might require plugins to handle users and permissions. When creating plugins for these sites, it is often important to

make sure you use the correct WordPress functions so that private information is not leaked to users without permission to see it.

This chapter gives you the tools necessary to work within the WordPress users, roles, and capabilities systems and shows you how they interact with one another within the WordPress environment.

WORKING WITH USERS

In WordPress, *users* are people who have registered a unique username for the site. The user has an account within that specific installation of WordPress. The term *user* should not be confused with a *visitor*, who is someone visiting or reading the website without an account. This chapter focuses on registered users.

Every WordPress installation has at minimum one user. This is the person who installed WordPress and initially set it up. Traditionally, this account was known as the *admin* user because older versions of WordPress automatically created an admin username. However, WordPress allows a different username upon registration today.

WARNING *Some older plugins relied on the existence of a user account called admin, which was wrong in the past and is wrong now. Never assume that a specific username is in use, and never assume that permissions are tied to a specific username.*

User Functions

WordPress has many functions for working with users. In this section, you'll learn how to use some of the most common functions needed for plugins.

WARNING *Many user functions are not loaded until after plugins are loaded, and the current user isn't authenticated until the init action hook (see [Chapter 5](#), “Hooks”). Using a user function before init in the WordPress flow will most likely cause a fatal error or incorrect data.*

is_user_logged_in()

The `is_user_logged_in()` function is a conditional function that allows you to check whether the current visitor is a logged-in user for the site. It returns a Boolean value of true or false based on whether the current user has an ID. The function is also pluggable, which means you can create a custom `is_user_logged_in()` function to overwrite WordPress' function completely.

In the following example, you'll display a basic message in the footer of the site based on whether the user is logged in. Logged-in users will receive one message, and those not logged in will get a different message.

```
<?php
add_action( 'wp_footer', 'pdev_maybe_logged_in_message' );

function pdev_maybe_logged_in_message() {

    if ( is_user_logged_in() ) {
        echo '<p>Welcome back! You are currently
logged in.</p>';
    } else {
        echo '<p>You are not logged into the site.
</p>';
    }
}
```

This function is important because it enables you to run specific code based on a user's logged-in status. While useful, you should not rely on it for permissions. You will likely need to check for specific capabilities for most use cases (see the “Roles and Capabilities” section of this chapter).

get_users()

The `get_users()` function enables you to query users from the database based on the arguments passed into the function through the `$args` parameter. The `$args` parameter is an array of arguments that you can define to limit the users returned by the function. The full list of arguments is available via the core documentation at

[developer.wordpress.org/reference/functions/get_users.](https://developer.wordpress.org/reference/functions/get_users/)

```
<?php
get_users( array $args = [] );
```

The function is technically a wrapper for the `WP_User_Query` class and returns an array of `WP_User` objects. If no users are found for the query, the return value will be an empty array.

Suppose you wanted to query all users assigned the `subscriber` role on the site. The `subscriber` role is one of several default roles that WordPress provides by default (see the “Roles and Capabilities” section of this chapter for more information on roles). The `get_users()` function accepts an argument named `'role'`, which can be a string for a single role or an array for multiple roles to query. The following code will query all subscribers and assign them to the `$subscribers` variable:

```
<?php
$subscribers = get_users( [
    'role' => 'subscriber'
] );
```

You could do anything with the array of user objects returned. For example, you could output all users’ avatar images by looping through them, as shown in the next snippet:

```
foreach ( $subscribers as $user ) {
    echo get_avatar( $user );
}
```

count_users

The `count_users()` function allows you to count the users of the site. It keeps track of the count of all users and the number of users for each role. Counting users is an intensive operation and should be used only when necessary. It should never be run on every page request.

```
<?php
count_users( string $strategy = 'time', int|null $site_id =
null );
```

The first parameter, `$strategy`, can be set to either `time` or `memory`. The `time` option, the default, is CPU-intensive and can handle around 10^7 users. The `memory` value is memory-intensive and can handle around 10^5 users. The second parameter, `$site_id`, allows you to count the users of a specific site on a multisite setup. It will default to the currently viewed site.

Imagine that you wanted to output a basic message with the total user count; you could do so by accessing the `total_users` value returned from `count_users()`.

```
<?php
// Get the user count.
$count = count_users();

// Output the total user count.
printf(
    '<p>This site has %s users.</p>',
    absint( $count['total_users'] )
);
```

You could also create a list of counts by user role via the `avail_roles` value from the function's returned array.

```
<?php
// Get the user count.
$count = count_users();

echo '<ul>';

// Output each role and its number of users.
foreach ( $count['avail_roles'] as $role => $user_count ) {

    printf(
        '<li>%1$s: %2$s</li>',
        esc_html( $role ),
        absint( $user_count )
    );
}

echo '</ul>';
```

Creating, Updating, and Deleting Users

The WordPress admin has built-in screens for creating, updating, and deleting users that most end users will use to administrate their site. However, as a plugin author, you may find yourself in scenarios where you need to create a plugin that handles these things outside the normal interface. Or, you may even want to try your hand at creating an even better interface.

The following are some examples of reasons why you might need to code a plugin to handle this:

- A client needs to import thousands of users from a different system into WordPress, and creating these users individually would be out of the question.
- You're building a social networking plugin that needs a frontend interface for registering user accounts.
- You're building a plugin that enables administrators to bulk edit/update various forms of user data in a quick and efficient manner.
- You need to create a sidebar widget or other frontend element that allows user creation.

As you should be able to see, there are many reasons for stepping outside the standard WordPress interface for handling users. While you will not be using the standard interface, you will be using standard functions that WordPress has provided for doing these types of things. These are the same functions that WordPress itself uses behind its user-related interfaces.

Creating a New User

When you need to create a new user in the database, WordPress provides the `wp_insert_user()` function. Technically, it can also handle updating a currently registered user if a user ID is passed in via the `ID` argument.

The function accepts a single parameter of `$userdata`. This is an array of arguments that match up to values in the `$wpdb->users` and `$wpdb->usermeta` tables in the database for the specific user.

The most important arguments for the `$userdata` parameter are as follows:

- `ID`: A current user's ID. You should use this only if you're updating a user. WordPress automatically creates new user IDs.
- `user_pass`: The password in plain text for the new user account.
- `user_login`: This is the “username” for the user. This is a required argument and returns an error if not unique.

- `user_nicename`: An alternative name to use in things such as permalinks to user archives. This defaults to the `user_login` argument.
- `user_url`: A link to the user's personal website.
- `user_email`: The email address of the user. This is a required argument and returns an error if not given or if the email address is already in use.
- `display_name`: The name to display for the user. This defaults to the `user_login` argument.
- `nickname`: A nickname for the user. This defaults to the `user_login` argument.
- `first_name`: The first name of the user.
- `last_name`: The last name (surname) of the user.
- `description`: A biographical information argument that describes the user.
- `user_registered`: The date and time of the user registration. WordPress automatically sets this to the current date and time if no argument is given.

You can view the full list of arguments via the `wp_insert_user()` function documentation at developer.wordpress.org/reference/functions/wp_insert_user/.

Now, you can take these arguments and create a new user with the `wp_insert_user()` function. You can mix and match the preceding arguments, but make sure you use the required arguments (`user_login`, `user_pass`, and `user_email`). The user you create next has the “editor” role and username of `johndoe`. You also make sure the user was created by displaying a WordPress-generated error message in the instance that something went wrong.

Create a new plugin file named `plugin-insert-user.php` and use the following code to create a plugin that will insert a new user into the database using the `wp_insert_user()` function:

```
<?php
/**
```

```

* Plugin Name: Insert User
* Plugin URI: http://example.com/
* Description: A plugin that inserts a "John Doe" user.
* Author: WROX
* Author URI: http://wrox.com
*/
add_action( 'init', 'pdev_insert_user' );

function pdev_insert_user() {

    // Bail if the user already exists.
    if ( username_exists( 'johndoe' ) ) {
        return;
    }

    // Create new user.
    $user = wp_insert_user( [
        'user_login'    => 'johndoe',
        'user_email'    => 'john@example.com',
        'user_pass'     => '123456789',
        'user_url'      => 'https://wordpress.org',
        'display_name'  => 'John Doe',
        'role'          => 'editor',
        'description'   => 'Loves to publish books on
WordPress!'
    ] );

    // If the user wasn't created, display error message.
    if ( is_wp_error( $user ) ) {
        echo $user->get_error_message();
    }
}

```

You may have taken notice of the use of the `username_exists()` function in the preceding code. It is a quick conditional function for checking whether a specific username is already registered with WordPress. It accepts a single parameter `$username`, which should be the username you are checking against. The code checks whether the username exists before attempting to insert a new user with that name.

The `wp_insert_user()` function is not the only method of creating a new user with WordPress. The `wp_create_user()` function is a quick wrapper function for `wp_insert_user()` that is available when you merely need to register a user with a new username, password, and email address.

```
<?php
wp_create_user( string $username, string $password, string
$email = '' );
```

Create a new plugin file named `plugin-create-user.php`. Now create a new user as you did in the previous plugin with a `janedoe` username. It should look similar to the following:

```
<?php
/**
 * Plugin Name: Create User
 * Plugin URI: http://example.com/
 * Description: A plugin that inserts a "Jane Doe" user.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'init', 'pdev_create_user' );

function pdev_create_user() {

    // Bail if the user already exists.
    if ( username_exists( 'janedoe' ) ) {
        return;
    }

    // Create new user.
    $user = wp_create_user(
        'janedoe',
        '123456789',
        'jane@example.com'
    );

    // If the user wasn't created, display error message.
    if ( is_wp_error( $user ) ) {
        echo $user->get_error_message();
    }
}
```

As you can see, `wp_insert_user()` and `wp_create_user()` are almost the same. Use the first when you need to insert more user data. Use the latter when you need to simply input the required username, password, and email parameters necessary for creating a unique user.

Updating an Existing User

As discussed in the previous section on creating new users, the `wp_insert_user()` function can be used to update an existing user by passing in a user ID via the `ID` argument. However, `wp_update_user()` should almost always be used instead.

When updating a user's password with `wp_update_user()`, the user's cookies will automatically be reset. The function will also send an email to the user to let them know their password has changed.

```
<?php
wp_update_user( array|object|WP_User $userdata );
```

This function takes in a single parameter called `$userdata`, which accepts all the same arguments covered in the section on `wp_insert_user()`. However, the `ID` argument is required for updating the user. If the `ID` argument isn't present, a new user will be created.

In the next example, you use the `wp_update_user()` function to force the currently logged-in user to use the “fresh” color scheme in the admin. This can be useful for making sure all users have a consistent experience in the admin. Create a new file called `plugin-force-admin-color.php` and use the following code to update the user:

```
<?php
/**
 * Plugin Name: Force Admin Color
 * Plugin URI: http://example.com/
 * Description: Makes sure the current user has the "fresh"
color scheme.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'admin_init', 'pdev_force_admin_color' );

function pdev_force_admin_color() {

    // Get the current WP_User object.
    $user = wp_get_current_user();

    // Bail if no current user object.
    if ( empty( $user ) ) {
        return;
    }
```

```

    // Get user's admin color scheme.
    $color = get_user_meta( $user->ID, 'admin_color',
true );

    // If not the fresh color scheme, update it.
    if ( 'fresh' !== $color ) {

        wp_update_user( [
            'ID'          => $user->ID,
            'admin_color' => 'fresh'
        ] );
    }
}

```

Deleting an Existing User

If you need to delete an existing user via code, WordPress provides the `wp_delete_user()` function. The function is used for deleting individual users and reassigning their posts to an alternative user. This is a destructive action that cannot be undone without restoring from a database backup.

```

<?php
wp_delete_user( int $id, int $reassign = null );

```

The function returns `true` when the operation has completed. It accepts two parameters.

- `$id`: The ID of the user to delete.
- `$reassign`: The ID of the user to set as the author of posts in which the user you're deleting has published. If this parameter is not set, the posts and links will be deleted from the database.

In this next example, you delete a user with the ID of `100` and reassign the user's posts and links to a user with the ID of `1`.

```

<?php
// Delete user 100 and assign posts to user 1.
wp_delete_user( 100, 1 );

```

WARNING *The wp_delete_user() function is available only in the WordPress admin. It is not loaded on the front end of the site. If you attempt to use it on the front end, the system will throw a fatal error.*

User Data

User data in WordPress is saved in two different tables in the database: \$wpdb->users and \$wpdb->usermeta. The users table primarily saves necessary information about the user. The usermeta table is for storing additional metadata about users. You sometimes need to load, create, update, and delete this data within your plugins.

When working with user data from the users table, you work with a few different values set for every user on the site.

- `ID`: The ID of the registered user.
- `user_login`: The login name (username).
- `user_pass`: The user's password. Note that you should never display this publicly.
- `user_nicename`: A “pretty” version of the user login that works in URLs.
- `user_url`: The website address of the user.
- `user_email`: The email address of the user. You should never display this publicly without the user's permission.
- `user_registered`: The date the user registered for the site.
- `display_name`: The name the user would like displayed.

Getting a User Object and Data

When accessing any user in a WordPress installation, nearly every function will utilize or return a `WP_User` object. This provides access to user data and other information about the user.

```
<?php
WP_User::__construct(
```

```

    int|string|stdClass|WP_User $id,
    string $name = '',
    int $site_id = ''
);

```

The constructor method of the class accepts three parameters.

- `$id`: The parameter can be a user ID, a `WP_User` object, or a user object directly from the database. Most often, you will pass in a user ID.
- `$name`: An optional username value for the user.
- `$site_id`: The site ID to pull the user from on multisite installations. It defaults to the current site.

Suppose you wanted to get the user object for the user with the ID of 1. You would use the following code:

```

<?php
$user = new WP_User( 1 );

```

The data available via the `$user` variable would look similar to the following:

```

object(WP_User) {
    'data' => object(stdClass) {
        'ID'          => 1,
        'user_login'   => 'example',
        'user_pass'    => '123456789', // hashed
        'user_nicename' => 'admin',
        'user_email'   => 'example@example.com',
        'user_url'     => 'example.com',
        'user_registered' => '2020-01-01 00:00:00',
        'user_activation_key' => '',
        'user_status'   => '0',
        'display_name'  => 'John Doe'
    },
    'ID'      => 1,
    'caps'    => [],
    'cap_key' => 'wp_capabilities',
    'roles'   => [],
    'allcaps' => [],
    'filter'  => NULL
}

```

As you can see, you can easily access most of the information you need about a user account with a simple call to the `WP_User` class.

If you prefer to work with a function instead of the `WP_User` class to quickly get user data, WordPress provides a `get_userdata()` function. However, it accepts only a single parameter of `$id`, which must be a user ID. Let's take a look at a quick example of displaying a user's email address when using that function:

```
<?php
$user = get_userdata( 100 );

echo $user->user_email;

// Prints:
// example@example.com
```

Getting the Current User Object

At times, you will need to get the current user object, which will provide some user data or give you an ID to fetch metadata later. The `wp_get_current_user()` function gets the user data from the `users` table for the currently logged-in user and returns it in the form of a `WP_User` object. It can be useful for displaying information when a specific user is logged into the site. The function accepts no parameters.

In the next example, you display a welcome message based on the user's display name in the WordPress admin footer.

```
<?php
add_action( 'in_admin_footer', 'pdev_user_welcome_message'
);

function pdev_user_welcome_message() {

    // Get current user object.
    $user = wp_get_current_user();

    // Display welcome message to user.
    printf(
        'Hello, %s.<br/>',
        esc_html( $user->display_name )
    );
}
```

Getting User Post Counts

If you need to count the number of posts a single user has written, use the `count_user_posts()` function.

```
<?php
count_user_posts(
    int $userid,
    array|string $post_type = 'post',
    bool $public_only = false
);
```

The function accepts three parameters.

- `$userid`: The ID of the user to count posts for.
- `$post_type`: A specific post type to count. You can pass in a string with a single post type name or an array of multiple post type names.
- `$public_only`: Whether to count only the posts that are public.

Suppose you have a rating system for users depending on the number of blog posts each user has written. You want to give a user a “silver” rating for writing 25 posts or a “gold” rating for writing 50 posts. To do this, you need to execute the code only when a post is saved. Create a new plugin file called `plugin-user-ratings.php` and use the following code to update the logged-in user's rating. Remember, the user's rating is saved as user meta only when a post is saved. (See the “User Metadata” section for details on how to handle metadata.)

```
<?php
/**
 * Plugin Name: User Ratings
 * Plugin URI: http://example.com/
 * Description: Updates user rating based on number of
 * posts.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'save_post', 'pdev_add_user_rating' );

pdev_add_user_rating() {
    // Get the current user object.
```

```

$user = wp_get_current_user();

// Get the user's current rating.
$rating = get_user_meta( $user->ID, 'user_rating',
true );

// Bail if user already has gold rating.
if ( 'gold' === $rating ) {
    return;
}

// Get the user's post count.
$posts = count_user_posts( $user->ID );

// Update the user's rating based on number of posts.
if ( 50 <= $posts ) {
    update_user_meta( $user->ID, 'user_rating',
'gold' );
} elseif ( 25 <= $posts ) {
    update_user_meta( $user->ID, 'user_rating',
'silver' );
}
}

```

Any time you need post counts from multiple users, you should use the `count_many_users_posts()` function instead of using the `count_user_posts()` function multiple times so that you're not querying the database multiple times.

```

<?php
count_many_users_posts(
    array $users,
    string|array $post_type = 'post',
    bool $public_only = false
);

```

The two functions' parameters are nearly identical. The big difference is that `count_many_users_posts()` expects an array of user IDs for the `$users` parameter as opposed to a single ID.

Suppose you had three specific user IDs (100, 200, and 300) that you wanted to list post counts for. Use the following code to create an HTML list of those post counts by user ID:

```

<?php
$users_counts = count_many_users_posts( [

```

```

100,
200,
300
] );
echo '<ul>';
foreach ( $users_counts as $user => $count ) {
    printf(
        '<li>The user with an ID of %1$s has written
%2$s posts.</li>',
        absint( $user ),
        absint( $count )
    );
}
echo '</ul>';

```

User Metadata

User metadata is data that is stored in the `$wpdb->usermeta` table in the database. It differs from user data stored in the `$wpdb->users` table because it is not required by WordPress. It is additional data that plugins can add to expand to their user experience.

This type of data can be anything you need to develop user-based settings for your plugin. This data is saved in key/value pairs; however, a single key may have multiple values. Meta keys are a way to represent the information provided by the meta values. Think of them as a human-readable ID that represents the information you're working with. Meta values are the pieces of data you retrieved based on the given meta key.

Metadata can literally be any type of data you want to save for the user. Some examples include the following:

- Twitter or Facebook account
- Location (country, city, state)
- Phone number
- Favorite book
- Personal settings for the site

- Private membership information

WARNING *Keep in mind that when saving sensitive information about users, you should make sure this data is not publicly displayed by your plugin.*

This section focuses on adding, displaying, updating, and deleting specific data for a user. This can enable you to see how the user meta functions work when manipulating data.

You work with a user ID of 100 and a meta key of `favorite_books`. This meta key saves the values of the user's three favorite books.

Adding User Metadata

To add new user metadata, use the `add_user_meta()` function. This function returns `true` when the data is successfully entered into the database and `false` when it fails.

```
<?php
add_user_meta(
    int $user_id,
    string $meta_key,
    mixed $meta_value,
    bool $unique = false
);
```

The function accepts four parameters.

- `$user_id`: The ID of the user to add metadata to.
- `$meta_key`: The metadata key in the database.
- `$meta_value`: A single value to add to pair with `$meta_key`.
- `$unique`: Whether the function should force a single row (`true`) in the `usermeta` table or create multiple rows for multiple meta values (`false`). This defaults to `false`.

Now, try adding three favorite books to a user with the ID of 100. These book titles are *WordPress Dev Champ*, *WordPress Lazy Coder*, and

WordPress the Hard Way.

```
<?php
add_user_meta( 100, 'favorite_books', 'WordPress Dev Champ',
false );
add_user_meta( 100, 'favorite_books', 'WordPress Lazy
Coder', false );
add_user_meta( 100, 'favorite_books', 'WordPress the Hard
Way', false );
```

The `$unique` parameter is set to `false`. This must be `false` to set multiple values for the same meta key, `favorite_books`. You would set this to `true` if the meta value should be a single value.

Retrieving User Metadata

Now that you've learned how to add custom user metadata, you might want to display it. The `get_user_meta()` function pulls the meta from the database based on the user ID and meta key.

```
<?php
get_user_meta( int $user_id, string $key = '', bool $single
= false );
```

The function accepts three parameters.

- `$user_id`: The ID of the user to get the metadata for.
- `$meta_key`: The metadata key to get the metadata value(s) for.
- `$single`: Whether to return an array of meta values (`false`) or a single value (`true`). This defaults to `false`.

To display an HTML list of the user's favorite books that you added with the `add_user_meta()` function, use the following code:

```
<?php
$books = get_user_meta( 100, 'favorite_books', false );

if ( $books ) {
    echo '<ul>';

    foreach ( $books as $book ) {
        printf(
            '<li>%s</li>',
            $book
        );
    }
}
```

```

        esc_html( $book )
    );
}

echo '</ul>';
}

```

If the `favorite_books` meta key had only a single meta value instead of multiple values, you wouldn't need to loop through an array. You could simply print the return value to the screen.

```

<?php
$book = get_user_meta( 100, 'favorite_books', true );
echo esc_html( $book );

```

Note that the preceding code set the third parameter, `$single`, to `true` for the `get_user_meta()` function. This means that it is looking for a single meta value.

Updating User Metadata

The `update_user_meta()` function enables you to update a single meta value whether there is a single or multiple values. Alternatively, you can completely overwrite all of the meta values if the meta key has multiple values. This function can also be used to insert new metadata if it doesn't already exist for the user, essentially performing the duty of the `add_user_meta()` function covered earlier.

```

<?php
update_user_meta(
    int $user_id,
    string $meta_key,
    mixed $meta_value,
    mixed $prev_value = ''
);

```

The function accepts four parameters.

- `$user_id`: The ID of the user you want to get metadata for.
- `$meta_key`: The metadata key to update meta values for.
- `$meta_value`: The new value for the meta key.

- `$prev_value`: The previous meta value to overwrite. If this is not set, all meta values will be overwritten with the single, new `$meta_value` parameter.

Suppose you want to change one of the user's favorite books, *WordPress Dev Champ*, to a new book, *WordPress Design Champ*. You need to set the `$meta_value` parameter to the new book name and the `$prev_value` parameter to the old book name.

```
<?php
update_user_meta(
    100,
    'favorite_books',
    'WordPress Design Champ',
    'WordPress Dev Champ'
);
```

If you want to overwrite all the favorite books with a single book, you can pass the `$meta_value` parameter and leave the `$prev_value` parameter empty.

```
<?php
update_user_meta( 100, 'favorite_books', 'WordPress Design
Champ' );
```

Deleting User Metadata

The `delete_user_meta()` function enables you to delete all the meta values and meta key or a single meta value for a given meta key.

```
<?php
delete_user_meta( int $user_id, string $meta_key, mixed
$meta_value = '' );
```

The function accepts three parameters.

- `$user_id`: The ID of the user to delete metadata for.
- `$meta_key`: The meta key to delete or the meta key to delete meta value(s) for.
- `$meta_value`: The specific meta value to delete. If this is left empty, all meta values and the meta key will be deleted for the user.

If you want to delete a single book from the user's list of favorite books, you need to set the `$meta_value` parameter to the name of the book. With the following code, you delete the *WordPress Lazy Coder* book from the user's favorite books.

```
<?php
delete_user_meta( 100, 'favorite_books', 'WordPress Lazy
Coder' );
```

If you want to delete all the user's favorite books, leave the `$meta_value` parameter empty.

```
<?php
delete_user_meta( 100, 'favorite_books' );
```

Creating a Plugin with User Metadata

Now that you have learned how to manipulate user metadata, it's time to use that knowledge for a practical use case. In many cases, custom user metadata your plugin might use needs to be set by the user from the user's profile page.

What you will do is build a plugin that adds an extra section to the user edit screen in the WordPress admin. This form will have a select box of the site's blog posts. The users can select one of these posts as their favorite.

The first step would be to create a new file in your plugin directory with a filename `plugin-user-favorite-post.php`. You would then need to create the plugin header and add the form to the user edit page.

```
<?php
/**
 * Plugin Name: User Favorite Post
 * Plugin URI: http://example.com/
 * Description: Allows users to select their favorite post.
 * Author: WROX
 * Author URI: http://wrox.com
 */

// Add the form to the user/profile admin screen.
add_action( 'show_user_profile',
'pdev_user_favorite_post_form' );
add_action( 'edit_user_profile',
'pdev_user_favorite_post_form' );
```

```

function pdev_user_favorite_post_form( $user ) {

    $favorite = get_user_meta( $user->ID, 'favorite_post',
true );

    $posts = get_posts( [ 'numberposts' => -1 ] ); ?>

    <h2>Favorites</h2>

    <table class="form-table">
        <tr>
            <th><label for="pdev-favorite-post">Favorite
Post</label></th>

            <td>
                <select name="pdev_favorite_post"
id="pdev-favorite-post">
                    <option value="" <?php selected( '',
$favorite ); ?>>
                    </option>

                    <?php foreach ( $posts as $post ) {
                        printf(
                            '<option value="%s"
%s>%s</option>',
                            esc_attr( $post->ID ),
                            selected( $post->ID,
$favorite, false ),
                            esc_html( $post->post_title )
                        );
                    } ?>
                </select>
                <br/>
                <span class="description">Select your
favorite post.</span>
            </td>
        </tr>
    </table></line><line xml:id="c09-line-0366"><![CDATA[<?
php }

```

This gives you an extra section on the user edit page, as shown in [Figure 9-1](#).

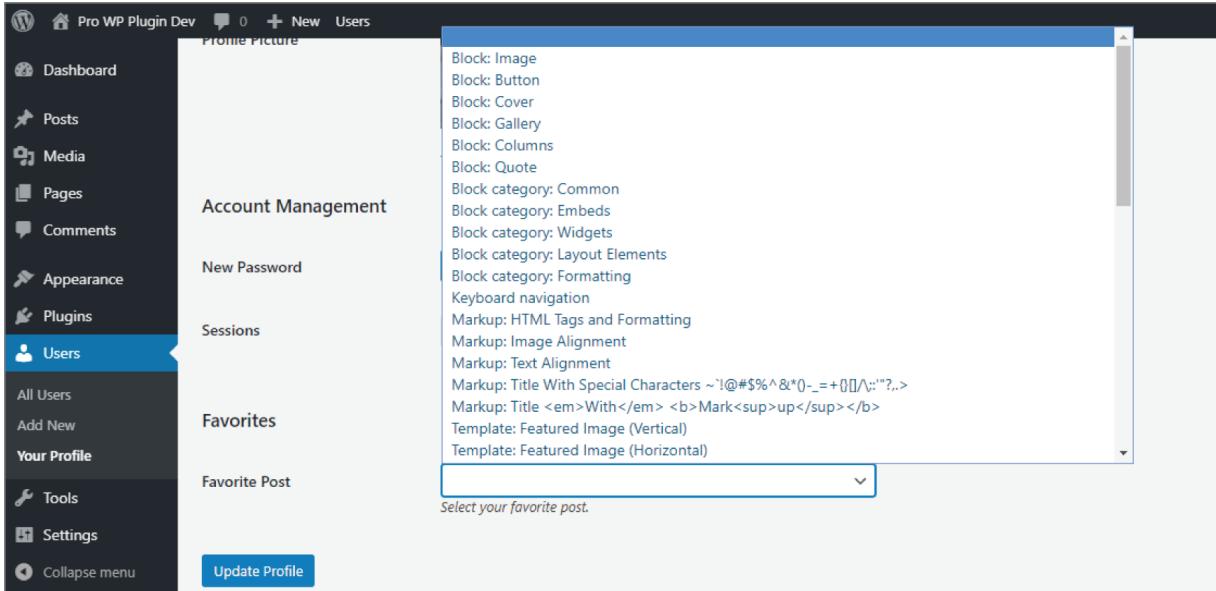


FIGURE 9-1: New form on the user edit page

The form is only displayed at this point. The next step is to save the user's favorite post as metadata. Append the following code to your `plugin-user-favorite-post.php` file:

```

// Add the update function to the user update hooks.
add_action( 'personal_options_update',
  'pdev_user_favorite_post_update' );
add_action( 'edit_user_profile_update',
  'pdev_user_favorite_post_update' );

function pdev_user_favorite_post_update( $user_id ) {

    // Bail if the current user cannot edit the user.
    if ( ! current_user_can( 'edit_user', $user_id ) ) {
        return;
    }

    // Get the existing favorite post if the value
    exists.
    // If no existing favorite post, value is empty
    string.
    $old_favorite = get_user_meta( $user_id,
        'favorite_post', true );

    // Sanitize to only accept numbers since it's a post
    ID.
    $new_favorite = preg_replace(
        "/[^0-9]/",

```

```

        '',
        $_POST['pdev_favorite_post']
);

// Update the user's favorite post.
update_user_meta( $user_id, 'favorite_post',
$favorite_post );

// If there's an old value but not a new value,
// delete old value.
if ( ! $new_favorite && $old_favorite ) {
    delete_user_meta( $user_id, 'favorite_post' );

    // If the new value doesn't match the new value,
    // add/update.
} elseif ( $new_value !== $old_value ) {
    update_user_meta( $user_id, 'favorite_post',
$new_value );
}
}

```

ROLES AND CAPABILITIES

WordPress provides a flexible user system, as you have seen in the previous half of this chapter. However, this system represents data about individual users. Alone, it does not define what a user can do within the site. For that, WordPress has a roles and capabilities system, which enables complete control over what permissions users have.

Roles are what define users in the WordPress system. More precisely, they grant users a set of permissions called *capabilities*.

What Are Roles and Capabilities?

Roles are what users are grouped by in WordPress. They're a way of giving a label to various sets of users. One of the most important things to note is that roles are not hierarchical. For example, an administrator is not necessarily “higher” than an editor. Roles are merely defined by what the role can and can't do. It's a permissions system.

Capabilities are that permissions system. Roles are assigned capabilities that define what that role can or can't do. These can be set up with the WordPress defaults or be completely custom, depending on the site. As a

plugin developer, you can't make assumptions about what certain roles have permission to do.

Imagine having to define what individual users could do on a site with thousands of users, each user having a custom set of permissions. It would be nearly impossible to maintain. Roles enable you to group users into distinguishable sets, each group with its own permissions.

Understanding how users, roles, and capabilities work and their relationship to one another is an important aspect of plugin development.

- **Users** are registered accounts on a site. Each user's role determines what the user can do on a site.
- **Roles** are sets of capabilities assigned to users. Multiple roles may be given to users, although this feature is not exposed in the WordPress interface by default.
- **Capabilities** are the permissions for roles, which are extended to the users of that role.

In general, most plugins will not need to know what roles users have. Most plugins work directly with capabilities because they are what define whether a user has permission to perform a task within the site.

WARNING *A common mistake many plugin authors make is to check a user's role before executing code. There is rarely a good reason to do this. Your plugin should check for a capability because capabilities determine a user's permission to do something on the site.*

Default Roles

WordPress ships with five default roles, which work well for most installs. Although roles are not hierarchical, the default roles do set up what appears to be a hierarchical system.

- **Administrator:** Has control over everything on the site
- **Editor:** Has publishing permission and editing access over everyone's posts

- **Author:** Has publishing access and can edit its own posts but doesn't have control over others' content
- **Contributor:** Can submit posts but not publish them
- **Subscriber:** Enables access to edit its user profile and the WordPress dashboard

This list is a general overview of the default roles. Even though these are the defaults, plugin users can install role management plugins that enable them to change what each role can do.

NOTE *On multisite installations, there is a Super Admin role that has global control over the network and all sites within the network. Super Admins bypass the capabilities system that other roles follow.*

Custom Roles

WordPress allows custom roles, which can be created by plugins. There is no user interface in WordPress for creating custom roles; however, several role management plugins exist that enable users to create new roles for their site.

Therefore, as a plugin developer, you can never know exactly what roles exist or might exist for a site unless you have direct access to the install, such as when doing client work. Keep this in mind when developing your plugins.

You may be faced with the task of creating custom roles in your plugins as well. Imagine that you were creating a plugin that implemented a forum within a WordPress install. Because WordPress would not know how to manage a forum, you would have to define these custom roles within your plugin. Some example roles for a forum might include the following:

- Forum Administrator
- Forum Moderator
- Forum Member
- Forum Banned

See the “Customizing Roles” section later in this chapter for more details on creating custom roles.

LIMITING ACCESS

“Limiting access” is a way to describe the process of working with WordPress capabilities to see whether a user has permission to access something or perform a specific task within a WordPress install.

By default, most access in WordPress is restricted within the admin. The admin can expose potentially vital information about a site, so making sure only users with the correct permissions have access to particular parts of it is important. WordPress can handle this when it needs to. Security issues arise when your plugin doesn't take into account a user's capabilities.

WordPress has two types of capabilities.

- **Primitive capabilities:** These capabilities are given to specific roles. Any capabilities for the role are extended to that role's users.
- **Meta capabilities:** These capabilities are not assigned to a role. They are checked against a specific object in WordPress, such as a post, user, or taxonomy term. Meta caps are then “mapped” to one or more primitive caps.

A good way to differentiate between the two is to consider this example. User A has the `edit_posts` capability (given to his role). This capability enables the user to edit posts. However, this user should not be able to edit User B's posts based on that capability. That's where meta capabilities come into play. If User A is trying to edit User B's post, the `edit_post` meta capability is called, but it's not the actual capability checked for. The `map_meta_cap()` function in WordPress decides whether the user can edit the post by returning an array of primitive capabilities to check against based on the user and post.

When adding settings pages for your plugin, you check for a capability. However, WordPress handles the capability checks for you, so you do not need to worry about limiting access to those pages with custom code. See

[Chapter 3](#), “Dashboard and Settings,” for more information on adding settings pages.

This chapter reviews only a few of the possible capabilities. For an in-depth list of the available capabilities, reference the Roles and Capabilities support page: wordpress.org/support/article/roles-and-capabilities/.

Checking User Permissions

When you check a user's permissions, you are checking whether a user's role has been granted a specific capability or whether the user is given a meta capability for a specific object. You may also check against a meta capability at times, which will be mapped to one or more of the existing primitive capabilities.

When checking whether a user has permission to do something, you almost exclusively use the `current_user_can()` function. It checks whether the currently logged-in user has permission to perform a given capability and returns `true` or `false`.

```
<?php
current_user_can( string $capability, mixed $args );
```

The function accepts two parameters.

- `$capability`: A single capability to check against a user's role.
- `$args`: Extra arguments to pass into the check. This is usually an object ID (like a post ID) when checking whether the user has a meta capability.

You will most likely use this function when checking for permissions within your plugin. You can use it to check for default WordPress capabilities or custom capabilities implemented by your plugin.

Suppose you wanted to check whether a user has permission to edit posts before creating a link to the posts page in the admin on the frontend of the site. You would use the `current_user_can()` function and the `edit_posts` capability.

```
<?php
if ( current_user_can( 'edit_posts' ) ) {
    printf(
        '<a href="%s">Edit Posts</a>',
        esc_url( admin_url( 'edit.php' ) )
    );
}
```

If you want to check for a meta capability, you would use the same technique. However, you need to insert the second parameter of the object ID into the `current_user_can()` function. For example, suppose you want to save some post metadata for a post with the ID of 100 but need to check whether the user can edit the post before updating the metadata (see [Chapter 8](#), “Content”).

```
<?php
$post_id = 100;

if ( current_user_can( 'edit_post', $post_id ) ) {
    update_post_meta( $post_id, 'pdev_example', 'Example'
);
}
```

WARNING *Your plugin should not check for permissions based on role. Remember, roles are not hierarchical, so you cannot assume a role has permission to perform a specific task. Always check for permission by capability.*

The `current_user_can()` function is actually a wrapper for `user_can()`. The difference is that `user_can()` accepts a first parameter of a user ID or user object.

```
<?php
user_can( int|WP_User $user, string $capability, mixed $args
);
```

You can check whether a specific post author has permission via the `author_can()` function.

```
<?php
author_can( int|WP_Post $post, string $capability, mixed
$args );
```

WordPress also provides the `current_user_can_for_blog()` function to check against a capability for a specific site in multisite setups.

```
<?php
current_user_can_for_blog( int $blog_id, string $capability,
mixed $args );
```

All of the available “user can” functions essentially do the same thing and check a specific user's permission. Take a quick look over how each of these permission-checking functions works.

```
<?php
$user_id = 1234;
$post_id = 9999;
$blog_id = 5555;

if ( user_can( $user_id, 'manage_options' ) ) {
    // Execute code if the user can manage options.
}

if ( current_user_can( 'edit_pages' ) ) {
    // Execute code if the current user can edit pages.
}

if ( author_can( $post_id, 'publish_posts' ) ) {
    // Execute code if the post author can publish posts.
}

if ( current_user_can_for_blog( $blog_id,
'edit_theme_options' ) ) {
    // Execute code if current user can edit theme
options for blog.
}
```

Each of these functions should be in your developer toolbox, but you will almost always use `current_user_can()`. The others will depend on the specific use case.

Is the User an Admin?

Sometimes, your plugin might need to check whether a user is an admin on the site. This can be confusing as a plugin developer because roles are not hierarchical. Remember, users with the “administrator” role are not always in full control of the site. However, they will be in most cases.

Generally, you would not check whether a user is an admin. It's nearly always better to check for a capability based on the specific task the user might be performing.

Determining admin status can be dangerous territory without understanding how roles and capabilities work because there is no single capability that defines a user as an admin. WordPress does provide a function for checking whether a user is an admin, but it should be used only if a specific capability isn't evident.

The `is_super_admin()` function was added as part of the multisite package of features, but it works for single-site installs of WordPress as well. It gives an accurate view of who is an admin on the site. The function accepts a single parameter of `$user_id`, which is the ID of the user you want to check admin status for. It returns `true` if the user is an admin or `false` if the user is not an admin.

`is_super_admin()` works a bit differently for multisite installs and single-site installs. A super admin in a multisite setup has full control over the site. You can read more about how super admins work within a multisite environment in [Chapter 13](#), “Multisite.”

On single-site installations, the `is_super_admin()` function checks whether the current user has the `delete_users` capability because it is a capability that would essentially give the user the most power on the site. This could just as easily be done with the `current_user_can()` function. It almost seems silly to check whether someone is an admin by checking whether they have the `delete_users` capability, especially if the task to be performed has nothing to do with deleting users.

Suppose your plugin needed to check whether a user with the ID of 100 was an admin before performing a specific task. You would use the following code to handle that check:

```
<?php
if ( is_super_admin( 100 ) ) {
    echo 'The user with the ID of 100 is a super admin.';
}
```

WARNING *Now that you know how to check for an admin, don't do it, unless you have to. Seriously, check against capabilities unless there is absolutely no other option.*

CUSTOMIZING ROLES

WordPress roles are flexible, and plugins can bend them in any way that suits their purposes. This section covers customizing roles, creating new roles, and adding capabilities to new or existing roles.

You need to know that any changes you make to roles are saved in the database. If your plugin makes a change, this change will not undo itself when your plugin is deactivated or uninstalled. Your plugin should remove custom capabilities that it has added and remove any roles it adds if no users are assigned to that role on deletion.

Creating a Role

WordPress enables the creation of custom roles by plugins. The best time to create a new role with your plugin is on the activation hook for your plugin (see [Chapter 2](#), “Plugin Framework”). Role creation needs to be done only once, so this is a good hook to use because it's fired only when your plugin is activated.

The `add_role()` function enables plugin developers to easily add new roles. The function returns a role object if a new role was successfully added and `null` if the role already exists.

```
<?php
add_role( string $role, string $display_name, array
$capabilities = [] );
```

The function accepts three parameters.

- `$role`: The name of the role to add. This should act as a key and contain only alphanumeric characters or underscores.
- `$display_name`: The label for the role. This is the name used for the role in public-facing areas.

- `$capabilities`: An array of capabilities to assign to the role. Capabilities can also be added or removed later. Capabilities should be the array keys. The array values should be either `true` to grant a capability to the role or `false` to explicitly deny a capability to the role.

Now go back to the previous forum example from the “Custom Roles” section. The section outlined four roles that a forum plugin might define.

- Forum Administrator
- Forum Moderator
- Forum Member
- Forum Suspended

You also need to decide which capabilities to provide to each role. You will work with the following capabilities. A real-world forum plugin would likely have dozens of capabilities, but these should give you an idea of what could be done.

- `read`: A core WordPress capability that provides access to the Dashboard and Profile admin screens
- `create_forums`: A fictional capability for creating forums
- `create_threads`: A fictional capability for creating forum threads
- `moderate_forums`: A fictional capability for moderating forums

Now you must decide which roles get which capabilities. The following code is what the roles for a forum and their capabilities might look like:

```
<?php
// Add a forum administrator role.
add_role( 'forum_administrator', 'Forum Administrator', [
    'read'          => true,
    'create_forums' => true,
    'create_threads' => true,
    'moderate_forums' => true
] );
// Add a forum moderator role.
```

```

add_role( 'forum_moderator', 'Forum Moderator', [
    'read'          => true,
    'create_threads' => true,
    'moderate_forums' => true
] );

// Add a forum member role.
add_role( 'forum_member', 'Forum Member', [
    'read'          => true,
    'create_threads' => true
] );

// Add a banned forum role.
add_role( 'forum_banned', 'Forum Banned', [
    'read'          => true,
    'create_forums' => false,
    'create_threads' => false,
    'moderate_forums' => false
] );

```

Each of the added roles is fairly straightforward. You have created a *faux* hierarchical system for roles by granting “higher” roles more capabilities and “lower” roles fewer capabilities.

You should notice that the `forum_banned` role's capabilities are handled differently. The code explicitly sets the forum-related capabilities to `false`. This tells WordPress that you absolutely do not want the role to have these permissions, even if the user has another role that grants them.

WARNING *Note that explicitly denying a capability is not the same as a role not having a capability. A user can have capabilities from multiple roles. One role may be granted the capability, and another might not have it at all. However, an explicitly denied capability is meant to overrule a granted capability from another role.*

Deleting a Role

Deleting a role is as simple as adding a new role by using the correct WordPress function. However, if your plugin needs to delete a role, it should check that no users on the site have the given role before deleting it. Otherwise, you might potentially break the plugin user's custom setup.

The best time to delete a role is during the uninstall procedure for your plugin (see [Chapter 2](#)). It is also good practice to provide plugin settings to enable the user to decide whether they want to delete your plugin's roles (see [Chapter 3](#)).

The `remove_role()` function can remove a role from the list of saved roles in the database. It accepts a single parameter of `$role`, which is the name of the role (not the label or display name).

```
<?php
remove_role( string $role );
```

The function accepts a single parameter of the role name. It does not return a value. If you wanted to remove the forum roles added in the previous section of this chapter, you use the following code:

```
<?php
remove_role( 'forum_administrator' );
remove_role( 'forum_moderator' );
remove_role( 'forum_member' );
remove_role( 'forum_banned' );
```

Adding Capabilities to a Role

Like creating custom roles, you can create custom capabilities. You can also add WordPress capabilities to new or existing roles. A capability doesn't technically exist if it's not given to a role. Your plugin should add capabilities to a role only once. Most likely, it will run this code on the plugin activation hook (see [Chapter 2](#)).

To add a capability to a role, you must first get the role object, which requires the `get_role()` function. This function accepts a single parameter of `$role`, which is the name of the role. It returns a `WP_Role` object if a role is found or `null` if not. After you get the role object, you would use the `add_cap()` method to grant a capability to the role.

```
<?php
get_role( string $role );
```

Suppose you want to grant the default WordPress Contributor role the ability to publish posts. Using this code, you can make the change.

```
<?php
$contributor = get_role( 'contributor' );

if ( null !== $role ) {
    $role->add_cap( 'publish_posts', true );
}
```

If you wanted to instead explicitly deny the capability, you would change the second parameter of `add_cap()` to `false` instead of `true`.

```
<?php
$contributor = get_role( 'contributor' );

if ( null !== $role ) {
    $role->add_cap( 'publish_posts', false );
}
```

Removing Capabilities from a Role

WordPress has you covered when it comes to deleting capabilities from a role. Any time you create a plugin that assigns custom capabilities for use with just your plugin, your plugin needs to clean up after itself and remove the capabilities it added. Your plugin should remove capabilities from a role only once instead of on every page load. Most likely, it will run this code on the plugin uninstall (see [Chapter 2](#)).

Like with adding capabilities, you must first get the role you want to remove capabilities from using the `get_role()` function. The difference is that you would use the `remove_cap()` method for removing the capability from the role.

In the previous section, you added the `publish_posts` capability to the `contributor` role. In your plugin's uninstall method, you would remove this capability.

```
<?php
$contributor = get_role( 'contributor' );

if ( null !== $role ) {
    $role->remove_cap( 'publish_posts' );
}
```

A Custom Role and Capability Plugin

Now that you have learned how to create custom roles and capabilities, it's time to put this knowledge to the test. In this section, you create a plugin that puts it all together.

The forum plugin you've been keeping in mind throughout the roles section of this chapter will be your starting point. You will piece together the different parts from the "Creating a Role" and "Deleting a Role" sections to build a plugin that adds custom forum roles on plugin activation and deletes those roles on uninstall. You will also give the default WordPress Administrator role the same capabilities as the Forum Administrator role.

First, create a new plugin folder named `plugin-forum-roles`. Then, add the following code into a `plugin.php` file within the plugin:

```
<?php
/**
 * Plugin Name: Forum Roles
 * Plugin URI: http://example.com/
 * Description: Creates example roles for a forum.
 * Author: WROX
 * Author URI: http://wrox.com
 */

register_activation_hook( __FILE__,
'pdev_create_forum_roles' );

function pdev_create_forum_roles() {

    // Get the administrator role.
    $administrator = get_role( 'administrator' );

    // Add forum capabilities to the administrator role.
    $administrator->add_cap( 'create_forums' );
    $administrator->add_cap( 'create_threads' );
    $administrator->add_cap( 'moderate_forums' );

    // Add a forum administrator role.
    add_role( 'forum_administrator', 'Forum
Administrator', [
        'read'          => true,
        'create_forums' => true,
        'create_threads' => true,
        'moderate_forums' => true
    ] );
}
```

```

// Add a forum moderator role.
add_role( 'forum_moderator', 'Forum Moderator', [
    'read'          => true,
    'create_threads' => true,
    'moderate_forums' => true
] );

// Add a forum member role.
add_role( 'forum_member', 'Forum Member', [
    'read'          => true,
    'create_threads' => true
] );

// Add a banned forum role.
add_role( 'forum_banned', 'Forum Banned', [
    'read'          => true,
    'create_forums' => false,
    'create_threads' => false,
    'moderate_forums' => false
] );
}

```

Now create an `uninstall.php` file with the following code and place it within the plugin folder:

```

<?php

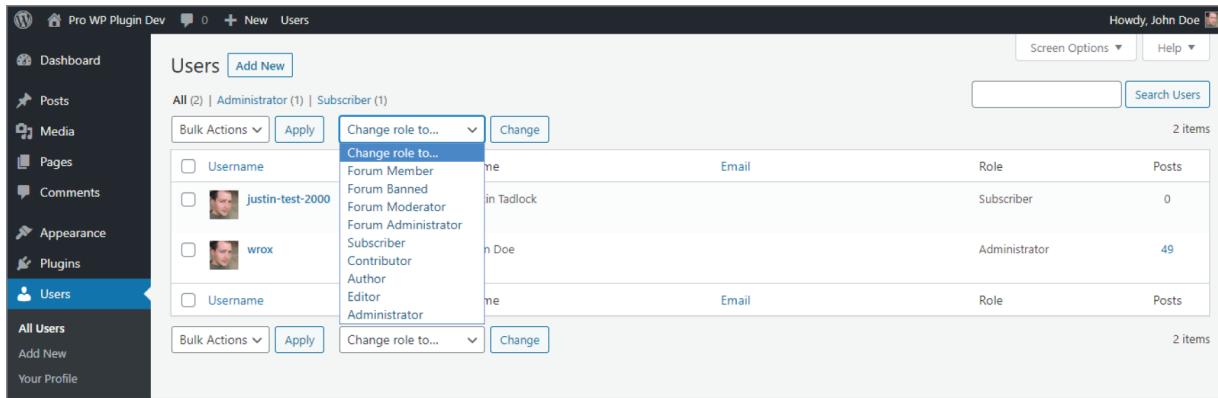
if ( ! defined( 'WP_UNINSTALL_PLUGIN' ) ) {
    wp_die( sprintf(
        '%s should only be called when uninstalling the
plugin.',

        __FILE__
    ) );
    exit;
}

remove_role( 'forum_administrator' );
remove_role( 'forum_moderator' );
remove_role( 'forum_member' );
remove_role( 'forum_banned' );

```

This plugin can make the four extra roles you created available on the Users screen in the admin, as shown in [Figure 9-2](#). Your plugin users could add the users they want to these forum-type roles.



The screenshot shows the WordPress 'Users' screen. On the left, a sidebar menu includes 'Dashboard', 'Posts', 'Media', 'Pages', 'Comments', 'Appearance', 'Plugins', and 'Users' (which is selected and highlighted in blue). Below these are 'All Users', 'Add New', and 'Your Profile'. The main content area is titled 'Users' with a 'Add New' button. It displays a table with two users: 'justin-test-2000' (Subscriber role) and 'wrox' (Administrator role). Above the table, there are 'Bulk Actions' and 'Apply' buttons, and a dropdown menu titled 'Change role to...' with options: 'Forum Member', 'Forum Banned', 'Forum Moderator', 'Forum Administrator', 'Subscriber', 'Contributor', 'Author', 'Editor', and 'Administrator'. The 'Administrator' option is highlighted. At the top right, there are 'Screen Options' and 'Help' buttons, and a search bar with the placeholder 'Search Users'.

FIGURE 9-2: New roles

SUMMARY

The user, role, and capability systems in WordPress are powerful and flexible, enabling you to build any type of plugin to manipulate how these systems work. Each section in this chapter has briefly touched on possibilities. The best thing you can do as a plugin developer is set up a test install and create several fictional test users. Then, take what you have learned throughout this chapter and start coding.

If you take away anything from this chapter, you need to remember how the relationship of users, roles, and capabilities works. Capabilities control permissions. Roles are given capabilities. Users are assigned roles, and each role's capabilities are extended to its users. Keeping this in mind when developing your plugins can make the development process much smoother.

10

Scheduled Tasks

WHAT'S IN THIS CHAPTER?

- Understanding cron in WordPress
- Managing recurring and single events
- Unscheduling cron events
- Viewing all scheduled cron tasks
- Creating custom intervals in cron
- Building out practical examples

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are at

www.wiley.com/go/prowordpressdev2e on the Downloads tab.

When developing plugins, you will sometimes need to execute functions on a schedule. WordPress provides such a task scheduler within the core code. In this chapter, you will learn how to schedule events, remove events from the schedule, and set custom schedule intervals. You will also learn from practice use cases.

WHAT IS CRON?

The term *cron* comes from the time-based task scheduler available in Unix systems. The cron system provided in core WordPress, while named the same, is not a true cron system in that sense. It is an event scheduler. However, it doesn't run on the system at all times. Events execute only when there's a page load.

Cron in WordPress is used to schedule jobs such as checking for new versions of WordPress, checking for plugin or theme updates, and

publishing scheduled posts.

How Is Cron Executed?

Unlike a Unix-based system, cron in WordPress does not constantly run on the server, looking for jobs to execute. WordPress, and thus its cron system, runs only when an admin or front-end page is loaded on your website.

When a page is visited, WordPress will automatically check if there are any cron tasks that it needs to execute. Because of how this system works, any visit to the website can trigger a scheduled task, regardless of whether it is from a human visitor or a bot.

The downside of how WordPress cron works is that it can never be 100 percent precise. For example, if you have a post scheduled to go live at 8 a.m. and no visitors come to your site until 10 a.m., then your post will not be published until 10 a.m.

As you can see, WordPress cron is more of a simulation of cron rather than a system-based cron. In general, it still serves most plugin needs.

SCHEDULING CRON EVENTS

You can schedule two types of cron events in WordPress: single and recurring. A single event only ever runs one time and will not run again until it is rescheduled. A recurring event is a task that runs on a schedule and repeats until it is unscheduled.

Scheduling a Recurring Event

When scheduling an event to execute with cron, what you are actually doing is creating a custom action hook (see [Chapter 5](#), “Hooks”). When scheduling the event with the cron scheduler, that hook is what gets registered with WordPress. When an event comes up in the schedule, any actions (functions) attached to that hook will be executed.

WordPress provides the `wp_schedule_event()` function for scheduling custom events.

```
<?php
wp_schedule_event(
```

```
    int $timestamp,  
    string $recurrence,  
    string $hook,  
    array $args = []  
);
```

The function accepts four parameters.

- \$timestamp: A Unix timestamp for when you want the event to occur.
- \$recurrence: How often the events should recur. The default available options are 'hourly', 'twicedaily', and 'daily'.
- \$hook: The action hook name to execute for the event.
- \$args: An array of arguments to pass to the hook's callback function.

Now you will build an example plugin to learn the basics of how to schedule tasks. First, create a registration callback for your plugin and schedule an hourly event during the callback.

```
<?php  
register_activation_hook( __FILE__, 'pdev_cron_activation'  
);  
  
function pdev_cron_activation() {  
  
    $args = [  
        'example@example.com'  
    ];  
  
    if ( ! wp_next_scheduled( 'pdev_hourly_email', $args  
 ) ) {  
  
        wp_schedule_event( time(), 'hourly',  
        'pdev_hourly_email', $args );  
    }  
}
```

If you look closely at the code, you can see a couple of things happening in the `pdev_cron_activation()` registration function. The first thing is the call to `wp_next_scheduled()`, which is a dual-purpose function. Prior to scheduling an event, you should always check that it isn't already registered.

The function's first purpose is to check whether an event is scheduled for the custom hook (`pdev_hourly_email`) that you have registered. If no event is scheduled, the function returns `false`. If the event is registered, the function will return the Unix timestamp of the next event in the schedule.

The next important item to note is that the `$hook` and `$args` parameters passed into `wp_next_scheduled()` and `wp_schedule_event()` must match exactly. The `$args` parameter is used to identify the event when there are multiple events on the same hook.

The next step in building the plugin involves adding an action on the custom action hook you created by scheduling an event. This works like any other action hook. Your callback function simply needs to execute whatever code it needs. For this example, when you scheduled the event, you passed an array of a single parameter, which was an email address. You use that address to send off an email.

```
<?php
add_action( 'pdev_hourly_email', 'pdev_send_email' );

function pdev_send_email( $email ) {

    wp_mail(
        sanitize_email( $email ),
        'Reminder',
        'Hey, remember to do that important thing!'
    );
}
```

At this point, you have successfully built a cron task. Every hour, WordPress will fire off an email to remind you to do something.

Take a look at the entire code to see how it works as a plugin, shown here:

```
<?php
/**
 * Plugin Name: Cron Recurring Event
 * Plugin URI: http://example.com/
 * Description: Sends an email every hour.
 * Author: WROX
 * Author URI: http://wrox.com
 */

register_activation_hook( __FILE__, 'pdev_cron_activation'
```

```

);

function pdev_cron_activation() {
    $args = [
        'example@example.com'
    ];
    if ( ! wp_next_scheduled( 'pdev_hourly_email', $args ) ) {
        wp_schedule_event( time(), 'hourly',
        'pdev_hourly_email', $args );
    }
}

add_action( 'pdev_hourly_email', 'pdev_send_email' );

function pdev_send_email( $email ) {
    wp_mail(
        sanitize_email( $email ),
        'Reminder',
        'Hey, remember to do that important thing!'
    );
}

```

Scheduling a Single Event

Cron is most useful for recurring events. The power of a cron system is typically the ability to have events recur. You probably won't need to schedule single events often. However, WordPress provides the ability to schedule events that run once and are automatically removed from the cron schedule. To schedule a single event, use the `wp_schedule_single_event()` function.

```

<?php
wp_schedule_single_event( int $timestamp, string $hook,
array $args = [] );

```

The function accepts three parameters.

- `$timestamp`: A Unix timestamp for when you want the event to occur
- `$hook`: The action hook name to execute for the event

- `$args`: An array of arguments to pass to the hook's callback function

As you should notice, the parameters are similar to the `wp_schedule_event()` function with the difference being that `wp_schedule_single_event()` has no parameter for setting how often the event occurs. This is because the event runs only once at the scheduled time.

Now you will build a simple plugin that sends off the same email from the previous section on recurring events. Like before, you will start by registering an event when the plugin is activated.

```
<?php
register_activation_hook( __FILE__,
  'pdev_cron_single_activation' );

function pdev_cron_single_activation() {

  $args = [
    'example@example.com'
  ];

  if ( ! wp_next_scheduled( 'pdev_single_email', $args
) ) {

    wp_schedule_single_event( time() + 3600,
  'pdev_single_email', $args );
  }
}
```

Instead of using `time()` to set the timestamp, the example code uses `time() + 3600`, which is an hour in seconds. That schedules the event to happen one hour from the time the plugin is activated.

Since this is a single event, try sending an email to thank the user for using your plugin. You use an action hook in the same way that you would for a recurring event.

```
add_action( 'pdev_single_email', 'pdev_send_email_once' );

function pdev_send_email_once( $email ) {

  wp_mail(
    sanitize_email( $email ),
```

```

        'Plugin Name - Thanks',
        'Thank you for using my plugin! If you need help
with it, let me know.'
    );
}

```

Now take a look at the plugin's full source code.

```

<?php
/**
 * Plugin Name: Cron Single Event
 * Plugin URI: http://example.com/
 * Description: Sends a scheduled email once.
 * Author: WROX
 * Author URI: http://wrox.com
 */

register_activation_hook( __FILE__,
'pdev_cron_single_activation' );

function pdev_cron_single_activation() {

    $args = [
        'example@example.com'
    ];

    if ( ! wp_next_scheduled( 'pdev_single_email', $args
) ) {

        wp_schedule_single_event( time() + 3600,
'pdev_single_email', $args );
    }
}

add_action( 'pdev_single_email', 'pdev_send_email_once' );

function pdev_send_email_once( $email ) {

    wp_mail(
        sanitize_email( $email ),
        'Plugin Name - Thanks',
        'Thank you for using my plugin! If you need help
with it, let me know.'
    );
}

```

NOTE *Single cron events do not need to be unscheduled. WordPress will automatically unschedule the event after the event has run.*

Unscheduling an Event

When you schedule a cron task, WordPress stores that task in the `wp_options` table in the database. This means any scheduled tasks added by your plugin will not be removed when your plugin is deactivated or uninstalled. Because the task is still registered, WordPress can't execute it because the plugin code is no longer available.

As a plugin author, it is your responsibility to always clean up after yourself, which includes removing any events you have scheduled. To properly unschedule an event, use the `wp_unschedule_event()` WordPress function.

```
<?php
wp_unschedule_event( int $timestamp, string $hook, array
$args = [] );
```

This function accepts three parameters.

- `$timestamp`: A Unix timestamp that matches the scheduled event's timestamp
- `$hook`: The action hook name for the event
- `$args`: An array of arguments to pass to the original event

Imagine that you had a previously scheduled event registered that looked like the following example:

```
<?php
wp_schedule_event( time(), 'hourly', 'pdev_example_event' );
```

To remove this event, you need to get the timestamp via the `wp_next_scheduled()` function. If that function returns a timestamp, then you can unschedule the event via `wp_unschedule_event()`.

```
<?php
$timestamp = wp_next_scheduled( 'pdev_example_event' );
```

```

if ( $timestamp ) {
    wp_unschedule_event( $timestamp, 'pdev_example_event'
);
}

```

Now you will build a full plugin that schedules an event when the plugin is activated and unschedule that same event when the plugin is deactivated.

```

<?php
/**
 * Plugin Name: Cron Scheduled Event
 * Plugin URI: http://example.com/
 * Description: Example of scheduling and unscheduling an
event.
 * Author: WROX
 * Author URI: http://wrox.com
 */

register_activation_hook( __FILE__,
'pdev_cron_example_activation' );

function pdev_cron_activation() {

    if ( ! wp_next_scheduled( 'pdev_example_event' ) ) {
        wp_schedule_event( time(), 'hourly',
(pdev_example_event' );
    }
}

register_deactivation_hook( __FILE__,
'pdev_cron_example_deactivation' );

function pdev_cron_example_deactivation() {

    $timestamp = wp_next_scheduled( 'pdev_example_event'
);

    if ( $timestamp ) {
        wp_unschedule_event( $timestamp,
(pdev_example_event' );
    }
}

add_action( 'pdev_example_event', 'pdev_example_email' );

function pdev_example_email() {

```

```
        wp_mail(
            'example@example.com',
            'Reminder',
            'Hey, remember to do that important thing!'
        );
    }
}
```

Specifying Your Own Cron Intervals

WordPress has three default options for cron intervals: `hourly`, `twicedaily`, and `daily`. However, you are not limited to those intervals if you need to schedule tasks. To create a custom interval, create a custom filter for the `cron_schedules` filter hook.

```
<?php
add_filter( 'cron_schedules', 'pdev_cron_schedules' );

function pdev_cron_schedules( $schedules ) {

    $schedules['weekly'] = [
        'interval' => 604800,
        'display'  => 'Once Weekly'
    ];

    return $schedules;
}
```

The `cron_schedules` filter hook passes an array of schedules and expects an array to be returned. The `pdev_cron_schedules()` filter adds a custom filter to the array of schedules. The `$schedules` variable stores all the cron schedules available in WordPress. You can manipulate this array like any other array before returning it. However, you should avoid removing any of the default items in the array provided by WordPress because other plugins may rely on them.

The `$schedules` variable is an associative array that expects specific keys for each item in the array. The name of your custom cron interval should be the array key. In this case, it is `weekly`. You can now use the `weekly` option when scheduling events.

The value for the cron interval should be an array that contains two specific keys and their corresponding values. The first key, `interval`, must be an

integer that represents the number of seconds between intervals. The preceding code set this interval at 608400, which is a week in seconds. The second key, `display`, is a human-readable name for the interval, which may be displayed by cron-management plugins.

To use the custom interval, you can add it as the second parameter in `wp_schedule_event()`, as shown in the following code snippet:

```
<?php
wp_schedule_event( time(), 'weekly', 'pdev_custom_event' );
```

Viewing Scheduled Cron Events

WordPress does not have a built-in way to view cron jobs. The average user wouldn't understand cron, so it makes sense that it is not included in the default user interface. However, as a plugin developer, you can build an admin page that lists all scheduled events.

The first step to doing this is creating a new plugin folder named `plugin-scheduled-events` with a file named `plugin.php` that contains the following code:

```
<?php
/**
 * Plugin Name: View Scheduled Events
 * Plugin URI: http://example.com/
 * Description: View cron tasks via the admin.
 * Author: WROX
 * Author URI: http://wrox.com
 */

require_once plugin_dir_path( __FILE__ ) . 'View.php';

$pdev_scheduled = new \PDEV\ScheduledEvents\View();
$pdev_scheduled->boot();
```

This code is just a basic bootstrap file that launches the plugin and the `View` class. Now, create a `View.php` file in the same folder with an empty class.

```
<?php
namespace PDEV\ScheduledEvents;

class View {
```

```
        // Class methods go here.  
    }
```

Now that you have set up the basics of your plugin, you need to add `boot()` and `submenuPage()` methods. The first method “boots” the class by adding an action to the WordPress `admin_menu` hook. The second method is the callback function for the action hook, which adds a custom submenu page under the “Tools” screen in the WordPress admin.

```
public function boot() {  
    add_action( 'admin_menu', [ $this, 'submenuPage' ] );  
}  
  
public function submenuPage() {  
  
    add_submenu_page(  
        'tools.php',  
        'Scheduled Events',  
        'Scheduled Events',  
        'manage_options',  
        'pdev-scheduled-events',  
        [ $this, 'template' ]  
    );  
}
```

The next step of the process is adding a `template()` method, which is the callback defined as the last parameter of `add_submenu_page()` in the preceding code. This method will output the HTML for the scheduled events page in the admin.

The first thing you need to do in the `template()` method is set up some variables. First, you need to get the cron array stored in the WordPress database via the `_get_cron_array()` function. The second variable are the schedules (cron intervals) via `wp_get_schedules()`. You also need to set a variable with a date format, which will be used for creating a human-readable date and time.

```
<?php  
public function template() {  
  
    $cron      = _get_cron_array();  
    $schedules = wp_get_schedules();  
    $date_format = 'M j, Y @ G:i';  
?>
```

```
<!-- Place HTML code here. -->  
<?php }
```

After setting up the necessary variables, create the HTML wrapper, a page heading, and a table with a header and empty body. Use the following code to replace the placeholder HTML comment in the `template()` method:

```
<div class="wrap">  
    <h1 class="wp-heading-inline">Scheduled Events</h1>  
  
    <table class="widefat fixed striped">  
  
        <thead>  
            <th>Next Run (GMT/UTC)</th>  
            <th>Schedule</th>  
            <th>Hook</th>  
        </thead>  
        <tbody>  
        </tbody>  
    </table>  
</div>
```

From this point, you will build out the table body's rows and columns using the variables you gathered at the top of the `template()` method. Add the following code within the `<tbody>` element:

```
<?php foreach ( $cron as $timestamp => $hooks ) : ?>  
  
    <?php foreach ( (array) $hooks as $hook => $events ) : ?>  
    >  
  
        <?php foreach ( (array) $events as $event ) : ?>  
  
        <tr>  
            <td>  
                <?php echo date_i18n(  
                    $date_format,  
                    wp_next_scheduled( $hook )  
                ); ?>  
            </td>  
            <td>  
                <?php if ( $event['schedule'] ) : ?>  
                <?php echo esc_html(  
                    $schedules[ $event['schedule'] ]  
                ); ?>  
            </td>  
        </tr>
```

```

] ['display']
    );
    ?>
    <?php else : ?>
        Once
        <?php endif; ?>
    </td>
    <td>
        <code><?php echo esc_html( $hook ); ?>
</code>
    </td>
</tr>

<?php endforeach; ?>

<?php endforeach; ?>

<?php endforeach; ?>

```

Now take a closer look at the full `view.php` class file as it all comes together, shown here:

```

<?php
namespace PDEV\ScheduledEvents;

class View {

    public function boot() {
        add_action( 'admin_menu', [ $this, 'submenuPage' ] );
    }

    public function submenuPage() {

        add_submenu_page(
            'tools.php',
            'Scheduled Events',
            'Scheduled Events',
            'manage_options',
            'pdev-scheduled-events',
            [ $this, 'template' ]
        );
    }

    public function template() {

        $cron      = _get_cron_array();
        $schedules = wp_get_schedules();
        $date_format = 'M j, Y @ G:i';
    }
}

```



```

esc_html( $hook ); ?></code>
                </td>
            </tr>

        <?php endforeach; ?>

        <?php endforeach; ?>
    <?php endforeach; ?>
    </tbody>
</table>
</div>
<?php }

}

```

Now you've built a plugin for displaying scheduled events. Once activated on the website, you can find the Scheduled Events page under the Tools menu in the WordPress admin. It should look similar to [Figure 10-1](#).

Next Run (GMT/UTC)	Schedule	Hook
Oct 20, 2019 @ 19:56	Once Hourly	wp_privacy_delete_old_export_files
Oct 21, 2019 @ 5:56	Twice Daily	wp_update_plugins
Oct 21, 2019 @ 5:56	Twice Daily	wp_update_themes
Oct 21, 2019 @ 5:56	Twice Daily	wp_version_check
Oct 21, 2019 @ 17:56	Once Daily	recovery_mode_clean_expired_keys
Oct 21, 2019 @ 17:56	Once Daily	wp_scheduled_delete
Oct 21, 2019 @ 17:56	Once Daily	delete_expired_transients
Oct 21, 2019 @ 17:56	Once Daily	wp_scheduled_auto_draft_delete
Oct 20, 2019 @ 14:01	Once	publish_future_post

[FIGURE 10-1](#): Scheduled Events page

TRUE CRON

As discussed earlier in this chapter, the WordPress cron system is not a true cron because it runs based on page requests and not a timed interval via the server. However, you can set up a true cron if you need scheduled tasks to run precisely at the time they are scheduled.

Before setting up a true cron, you must disable cron in WordPress by adding the following code to your `wp-config.php` file:

```
define( 'DISABLE_WP_CRON', true );
```

This line disables WordPress from loading its `wp-cron.php` file, which is used to look for cron tasks to execute. Once you've disabled WordPress cron, you will need to set up another method to execute cron on the server. Such a system is outside the scope of this book.

One common method is using `wget` to load `wp-cron.php` on a schedule. If you are running a Linux-based server, a true cron will already exist. You can schedule tasks using the `crontab` command. If you are running a Windows-based server, you will need to install `wget`. The `wget` command to execute tasks would look like the following:

```
wget http://www.example.com/wp-cron.php
```

WordPress looks for all scheduled cron tasks and executes them as needed when `wget` requests `wp-cron.php`.

PRACTICAL USE

Now that you have an understanding of the foundational elements of cron, it is time to put that newfound knowledge to practical use. In this section, you will learn how to build plugins for real-world use cases.

The Blog Pester Plugin

Assume that you have a user who wants to write blog posts more regularly. This person has a hard time reminding themselves to log into their WordPress install and write a new blog post. They have commissioned you to write a plugin that will remind them to write something if there are no new posts every three days. This is a perfect use case for cron because it is based on a timed interval.

Create a new file called `plugin-blog-pester.php`. Then create an activation hook for the plugin, which will register the new cron task.

```
register_activation_hook( __FILE__, 'pdev_pester_activate' );
```

```

function pdev_pestter_activate() {

    if ( ! wp_next_scheduled( 'pdev_pestter_event' ) ) {
        wp_schedule_event( time(), 'daily',
'pdev_pestter_event' );
    }
}

```

The preceding code creates a hook called `pdev_pestter_event`. This hook is scheduled to fire once every day. You will later add a custom action to it to handle the check and send the reminder.

Remember that any recurring cron task that you schedule should always be removed when your plugin is no longer active. You will be a good citizen within the development community and remove the cron task with a custom deactivation hook so that WordPress will not look for it when running cron.

```

register_deactivation_hook( __FILE__,
'pdev_pestter_deactivate' );

function pdev_pestter_deactivate() {

    $timestamp = wp_next_scheduled( 'pdev_pestter_event'
);

    if ( false !== $timestamp ) {
        wp_unschedule_event( $timestamp,
'pdev_pestter_event' );
    }
}

```

With the basics out of the way, you can start building out the functionality for the custom plugin. What you need to do is add a custom callback function (action) to `pdev_pestter_event`. That function must first check to see whether there have been any posts published in the past three days. If not, the plugin must send an email to remind the site owner.

```

add_action( 'pdev_pestter_event', 'pdev_pestter_check' );

function pdev_pestter_check() {
    global $wpdb;

    // Query the latest published post date.
    $query = "SELECT post_date

```

```

        FROM $wpdb->posts
        WHERE post_status = 'publish'
        AND post_type = 'post'
        ORDER BY post_date
        DESC LIMIT 1";

$latest_post_date = $wpdb->get_var( $wpdb->prepare(
$query ) );

// Check if latest post is older than three days.
// If it is, send email reminder.
if ( strtotime( $latest_post_date ) <= strtotime( '-3
days' ) ) {

    $email    = 'example@example.com';
    $subject = 'Blog Reminder';
    $message = 'Hey! You have not written a blog
post in three days!';

    wp_mail( $email, $subject, $message );
}
}

```

The preceding code first runs a custom database query for the latest post date (see [Chapter 4](#), “Security and Performance,” to learn more about database queries). It then uses the PHP `strtotime()` function to compare dates. This function will accept any date or textual English date time. It then returns a Unix timestamp, which makes it easy to compare the two dates because they are in the same format. In the previous code, you are comparing the `$latest_post_date` variable from the database to the timestamp for three days ago, which is written as `'-3 days'`. If the latest post date is less than or equal to the value of `'-3 days'`, the post is older than three days. At that point, it is simply a matter of populating the email variables and firing off the reminder email via the `wp_mail()` function.

All that is left is filling in the plugin header. Review the following to see what the full plugin should look like:

```

<?php
/**
 * Plugin Name: Blog Pester
 * Plugin URI: http://example.com/
 * Description: Sends a reminder email if no posts have been
written in three days.
 * Author:      WROX

```

```

* Author URI:  http://wrox.com
*/
register_activation_hook( __FILE__, 'pdev_pest_activate' );
function pdev_pest_activate() {
    if ( ! wp_next_scheduled( 'pdev_pest_event' ) ) {
        wp_schedule_event( time(), 'daily',
'pdev_pest_event' );
    }
}

register_deactivation_hook( __FILE__,
'pdev_pest_deactivate' );
function pdev_pest_deactivate() {
    $timestamp = wp_next_scheduled( 'pdev_pest_event' );
    if ( false !== $timestamp ) {
        wp_unschedule_event( $timestamp,
'pdev_pest_event' );
    }
}

add_action( 'pdev_pest_event', 'pdev_pest_check' );
function pdev_pest_check() {
    global $wpdb;

    // Query the latest published post date.
    $query = "SELECT post_date
        FROM $wpdb->posts
        WHERE post_status = 'publish'
        AND post_type = 'post'
        ORDER BY post_date
        DESC LIMIT 1";

    $latest_post_date = $wpdb->get_var( $wpdb->prepare(
$query ) );

    // Check if latest post is older than three days.
    // If it is, send email reminder.
    if ( strtotime( $latest_post_date ) <= strtotime( '-3
days' ) ) {

```

```

        $email  = 'example@example.com';
        $subject = 'Blog Reminder';
        $message = 'Hey! You have not written a blog
post in three days!';

        wp_mail( $email, $subject, $message );
    }
}

```

Deleting Post Revisions Weekly

WordPress saves a post revision in the posts table in the database each time you save a post or page. This can quickly get out of hand and bloat the size of your database. It is a good idea to purge these revisions once in a while. What you will do is create a plugin that schedules a weekly cron task that deletes all post revisions from the database if those revisions are older than 30 days.

For this plugin, create a new `plugin-clean-revisions.php` file. Like in the previous plugin example, create both an activation function that will schedule a cron task and a deactivation function that will remove it.

```

register_activation_hook( __FILE__,
'pdev_clean_rev_activate' );

function pdev_clean_rev_activate() {

    if ( ! wp_next_scheduled( 'pdev_clean_rev_event' ) )
{
        wp_schedule_event( time(), 'weekly',
'pdev_clean_rev_event' );
    }
}

register_deactivation_hook( __FILE__,
'pdev_clean_rev_deactivate' );

function pdev_clean_rev_deactivate() {

    $timestamp = wp_next_scheduled(
'pdev_clean_rev_event' );

    if ( false !== $timestamp ) {
        wp_unschedule_event( $timestamp,

```

```

        'pdev_clean_rev_event' );
    }
}

```

One important thing to note is that the `pdev_clean_rev_event` hook will run once a week. Because `weekly` is not a value available in the default WordPress cron schedules array, you need to filter the schedules and add in the custom `weekly` option.

```

add_filter( 'cron_schedules',
'pdev_clean_rev_cron_schedules' );

function pdev_clean_rev_cron_schedules( $schedules ) {

    $schedules['weekly'] = [
        'interval' => 604800,
        'display'  => 'Once Weekly'
    ];

    return $schedules;
}

```

The preceding code merely tells WordPress that you want to run an event every 604800 seconds, once every week, via a custom filter on the `cron_schedules` filter hook. You can review custom schedules in the “Specifying Your Own Cron Intervals” section earlier in this chapter.

Now you must add a custom action to the `pdev_clean_rev_event` hook for it to fire each week.

```

add_action( 'pdev_clean_rev_event', 'pdev_clean_rev_delete' );

function pdev_clean_rev_delete() {
    global $wpdb;

    $sql = "DELETE a,b,c
            FROM $wpdb->posts array
            LEFT JOIN $wpdb->term_relationships b ON (a.ID
= b.object_id)
            LEFT JOIN $wpdb->postmeta c ON (a.ID =
c.post_id)
            WHERE a.post_type = 'revision'
            AND DATEDIFF( now(), a.post_modified )> 30";

```

```
        $wpdb->query( $wpdb->prepare( $sql ) );
    }
```

The \$sql variable stores the custom query to be executed. It joins the posts table with the term_relationships and postmeta tables. The reason for this is that data related to post revisions is stored in all three tables. When deleting the post revision, you must also delete any taxonomy term relationships and post metadata stored for each revision.

While there are no variables defined for this particular query, it is still good practice to wrap \$sql in \$wpdb->prepare() when running database queries. If you change the code in the future, it will already be in place. Essentially, it is something you should always do. Finally, you pass the SQL query into \$wpdb->query(), which runs the query and deletes all revisions older than 30 days along with their accompanying data.

NOTE *While unnecessary for SQL statements without variables, remember that it is good practice to always wrap them with \$wpdb->prepare(). This will ensure that any future additions of a variable are secure.*

Once you add your plugin header at the top of the file, you will have a fully functioning plugin. Now take a moment to review the full plugin code, shown here:

```
<?php
/**
 * Plugin Name: Clean Revisions
 * Plugin URI: http://example.com/
 * Description: Removes post revisions older than 30 days
every week.
 * Author: WROX
 * Author URI: http://wrox.com
 */

register_activation_hook( __FILE__,
'pdev_clean_rev_activate' );

function pdev_clean_rev_activate() {
    if ( ! wp_next_scheduled( 'pdev_clean_rev_event' ) )
{
```

```

        wp_schedule_event( time(), 'weekly',
'pdev_clean_rev_event' );
    }
}

register_deactivation_hook( __FILE__,
'pdev_clean_rev_deactivate' );

function pdev_clean_rev_deactivate() {

    $timestamp = wp_next_scheduled(
'pdev_clean_rev_event' );

    if ( false !== $timestamp ) {
        wp_unschedule_event( $timestamp,
'pdev_clean_rev_event' );
    }
}

add_filter( 'cron_schedules',
'pdev_clean_rev_cron_schedules' );

function pdev_clean_rev_cron_schedules( $schedules ) {

    $schedules['weekly'] = [
        'interval' => 604800,
        'display'  => 'Once Weekly'
    ];

    return $schedules;
}

add_action( 'pdev_clean_rev_event', 'pdev_clean_rev_delete'
);

function pdev_clean_rev_delete() {
    global $wpdb;

    $sql = "DELETE a,b,c
        FROM $wpdb->posts array
        LEFT JOIN $wpdb->term_relationships b ON (a.ID
= b.object_id)
        LEFT JOIN $wpdb->postmeta c ON (a.ID =
c.post_id)
        WHERE a.post_type = 'revision'
        AND DATEDIFF( now(), a.post_modified )> 30";
}

```

```
        $wpdb->query( $wpdb->prepare( $sql ) );
    }
```

The Delete Comments Plugin

Imagine that you wanted to build a plugin that would automatically delete spam or moderated comments from the database after they have been around for a while. You also want to release this plugin to the general public for others to use on their sites.

The previous two practical examples in this chapter primarily explored scheduled tasks in isolation. They work well as small, one-off plugins for a single website. However, in a plugin that you are building for public release for many users, you will want to include more advanced capabilities, such as adding custom plugin options that users can change.

Start by creating a new `plugin-delete-comments.php` file for your new plugin. You will build an entire plugin in this file.

Because this plugin has custom database options, the first thing you need to think about is what those options will be. By default, you want to delete old spam comments. However, users might want to delete old comments held in moderation. That's one option. You also think deleting comments older than 15 days is a good idea, but there is no need to limit to that number.

Providing an option to change the number of days is a good second option.

Your plugin will need to access these options multiple times. It is good practice to create a wrapper function that will return an array of all the options.

```
function pdev_delete_comments_options() {
    return get_option( 'pdev_delete_comments', [
        'status' => 'spam',
        'days'   => 15
    ] );
}
```

This code looks for an option in the database named `pdev_delete_comments`, which stores all of the plugin's options in a single array. It also passes along default values in case the option has not yet been stored in the database. The `status` option returns the default status (`spam`)

that you want to delete comments by. The days option returns the default number of days (15).

With the initial code for the options in place, it is time to register settings with WordPress by adding a callback to the `admin_init` action hook.

```
add_action( 'admin_init', 'pdev_delete_comments_init' );

function pdev_delete_comments_init() {

    // Register settings on the discussion screen.
    register_setting(
        'discussion',
        'pdev_delete_comments'
    );

    // Register comment status field.
    add_settings_field(
        'pdev_comment_status',
        'Comment Status to Delete',
        'pdev_comment_status_field',
        'discussion',
        'default'
    );

    // Register days field.
    add_settings_field(
        'pdev_comment_days',
        'Delete Comments Older Than',
        'pdev_comment_days_field',
        'discussion',
        'default'
    );

    // Schedule the cron event if not scheduled.
    if ( ! wp_next_scheduled(
        'pdev_delete_comments_event' ) ) {
        wp_schedule_event( time(), 'daily',
        'pdev_delete_comments_event' );
    }
}
```

The first thing the preceding code does is register a custom setting named `pdev_delete_comments` to the `discussion` page. The `discussion` string is a unique ID to let WordPress know that you want to add custom settings to the Discussion screen under the Settings menu in the WordPress admin.

Because the plugin that you are building is related to comments, this is the ideal place to add its settings. You can learn more about registering settings in [Chapter 3](#), “Dashboard and Settings.”

The next two blocks of code in the function register custom settings fields. This tells WordPress that the plugin has two callback functions, `pdev_comment_status_field()` and `pdev_comment_days_field()`, that will output the HTML on the settings screen.

The final code block in the function is where you get to start applying what you have learned in this chapter about cron. Like you should always do before registering a new cron event, check that it has not yet been scheduled. The previous code adds a hook named `pdev_delete_comments_event` that will fire once each day.

The next step in building the plugin is creating the callback functions for the settings fields.

```
function pdev_comment_status_field() {

    $options = pdev_delete_comments_options();
    $status  = $options['status']; ?>

    <select name="pdev_delete_comments[status]">
        <option value="spam" <?php selected( $status,
    'spam' ); ?>>
            Spam
        </option>
        <option value="moderated" <?php selected(
    $status, 'moderated' ); ?>>
            Moderated
        </option>
    </select>

<?php }

function pdev_comment_days_field() {

    $options = pdev_delete_comments_options();
    $days    = absint( $options['days'] );

    printf(
        '<input type="number"
name="pdev_delete_comments[days]" value="%s">',
        esc_attr( $days )
```

```
    );
}
```

The `pdev_comments_status_field()` function outputs a `<select>` element with an option to choose between spam or moderated comments. Take note of the `selected()` function, which is a core WordPress helper function for forms. It will automatically output the `selected` attribute by checking whether the `$status` variable is equal to the `<option>` field's value.

The `pdev_comment_days_field()` function outputs a basic number input. It uses `absint()` when defining the `$days` variable to make sure the number of days is a positive integer.

With this code in place, you should see the output on the `Settings ➔ Discussion` screen in the WordPress admin that is shown in [Figure 10-2](#).

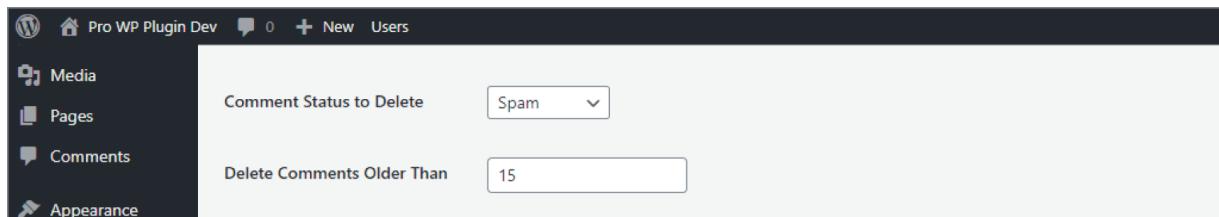


FIGURE 10-2: Number output

At this point, you need to add an action to the cron hook registered earlier. The action will handle deleting comments.

```
add_action( 'pdev_delete_comments_event',
'pdev_delete_comments_task' );

function pdev_delete_comments_task() {
    global $wpdb;

    $options = pdev_delete_comments_options();
    $status  = $options['status'];
    $days    = absint( $options['days'] );

    // Set default comment_approved value to spam.
    $comment_approved = 'spam';

    // If moderated status, WordPress stores this as '0'.
    if ( 'moderated' !== $status ) {
        $comment_approved = '0';
    }
}
```

```

        // Build and run the query to delete comments.
        $sql = "DELETE FROM $wpdb->comments
                WHERE ( comment_approved = '$comment_approved'
        )
                AND DATEDIFF( now(), comment_date ) > %d";
        $wpdb->query( $wpdb->prepare( $sql, $days ) );
    }
}

```

The previous code runs a database query to delete comments based on the options selected by the user from the Discussion admin page. It deletes comments based on the `comment_approved` field, which is the field in the `$wpdb->comments` table for storing the comment's status. It also checks how old the comment is in comparison to the moment it is run. If older than the number of days set by the user, the comment is deleted.

WordPress is more than 15 years old, which means it has some odd legacy bits in the system. That is why the `comment_approved` field for moderated comments in the database is stored as '`0`' while spam comments are stored as '`spam`'. Because of this, the preceding code creates a new variable named `$comment_approved` to define the proper value for the `comment_approved` database field.

One final step left. Don't forget one of the most crucial aspects of scheduling tasks in WordPress. Make sure to delete anything you have scheduled. Add in a deactivation hook that unschedules `pdev_delete_comments_event`.

```

register_deactivation_hook( __FILE__,
    'pdev_delete_comments_deactivate' );

function pdev_delete_comments_deactivate() {
    $timestamp = wp_next_scheduled(
    'pdev_delete_comments_event' );

    if ( false !== $timestamp ) {
        wp_unschedule_event( $timestamp,
    'pdev_delete_comments_event' );
    }
}

```

You have now built a comment-deletion plugin! Review the following code for a look at the plugin in its entirety:

```
<?php
/**
 * Plugin Name: Delete Comments
 * Plugin URI:  http://example.com/
 * Description: Deletes old spam or moderated comments on a
schedule.
 * Author:      WROX
 * Author URI:  http://wrox.com
 */

function pdev_delete_comments_options() {

    return get_option( 'pdev_delete_comments', [
        'status' => 'spam',
        'days'    => 15
    ] );
}

add_action( 'admin_init', 'pdev_delete_comments_init' );

function pdev_delete_comments_init() {

    // Register settings on the discussion screen.
    register_setting(
        'discussion',
        'pdev_delete_comments'
    );

    // Register comment status field.
    add_settings_field(
        'pdev_comment_status',
        'Comment Status to Delete',
        'pdev_comment_status_field',
        'discussion',
        'default'
    );

    // Register days field.
    add_settings_field(
        'pdev_comment_days',
        'Delete Comments Older Than',
        'pdev_comment_days_field',
        'discussion',
        'default'
    );

    // Schedule the cron event if not scheduled.
    if ( ! wp_next_scheduled(
```

```

'pdev_delete_comments_event' ) ) {
        wp_schedule_event( time(), 'daily',
'pdev_delete_comments_event' );
    }
}

function pdev_comment_status_field() {

    $options = pdev_delete_comments_options();
    $status  = $options['status']; ?>

        <select name="pdev_delete_comments[status]">
            <option value="spam" <?php selected( $status,
'spam' ); ?>>
                Spam
            </option>
            <option value="moderated" <?php selected(
$status, 'moderated' ); ?>>
                Moderated
            </option>
        </select>

<?php }

function pdev_comment_days_field() {

    $options = pdev_delete_comments_options();
    $days    = absint( $options['days'] );

    printf(
        '<input type="number"
name="pdev_delete_comments[days]" value="%s">',
        esc_attr( $days )
    );
}

add_action( 'pdev_delete_comments_event',
'pdev_delete_comments_task' );

function pdev_delete_comments_task() {
    global $wpdb;

    $options = pdev_delete_comments_options();
    $status  = $options['status'];
    $days    = absint( $options['days'] );

    // Set default comment_approved value to spam.
    $comment_approved = 'spam';
}

```

```

// If moderated status, WordPress stores this as '0'.
if ( 'moderated' !== $status ) {
    $comment_approved = '0';
}

// Build and run the query to delete comments.
$sql = "DELETE FROM $wpdb->comments
        WHERE ( comment_approved = '$comment_approved'
)
        AND DATEDIFF( now(), comment_date )> %d";

$wpdb->query( $wpdb->prepare( $sql, $days ) );
}

register_deactivation_hook( __FILE__,
'pdev_delete_comments_deactivate' );

function pdev_delete_comments_deactivate() {

    $timestamp = wp_next_scheduled(
'pdev_delete_comments_event' );

    if ( false !== $timestamp ) {
        wp_unschedule_event( $timestamp,
'pdev_delete_comments_event' );
    }
}

```

SUMMARY

Cron is a powerful tool that is often underused by plugin developers. It creates a lot of interesting possibilities. Understanding how cron works can allow you to build out more advanced features for your plugin users. Now it is time to start thinking about all the things you can do with it.

11

Internationalization

WHAT'S IN THIS CHAPTER?

- Understanding the description of internationalization and localization
- Determining the benefits of internationalizing plugins
- Preparing plugins for translation
- Using the WordPress internationalization functions
- Internationalizing JavaScript
- Using translation tools

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at
www.wiley.com/go/prowordpressdev2e on the Downloads tab.

Internationalization is the act of preparing your plugin for use in any number of languages. WordPress uses U.S. English as its default language, but it has a large community of users who don't read and write in English. This community pulls together to create translations of WordPress in languages used all around the world.

One of the goals of WordPress is to make it easy for people across the world to publish content. As a plugin developer, you can help democratize the publishing process for users of many different cultures. WordPress makes this easy for developers, so there are no development hurdles to cross when internationalizing your plugins.

INTERNATIONALIZATION AND LOCALIZATION

Using the built-in translation functions in WordPress, you can easily make your plugin available to a wide variety of people without any knowledge of your users' written languages. The process of translation is handled for you by WordPress if you follow a few simple steps during the development process.

Internationalization deals with making sure strings of text are wrapped in specific function calls. It is the practice of making text ready for localization. The shorthand term for internationalization is *i18n*, which you may encounter in some situations.

Localization is the process of translating text for a specific locale. WordPress handles the localization process by checking for specific translation files and performing the translation. It's a plugin's job to handle internationalization so that localization can take place. The shorthand term for localization is *L10n*.

WARNING *You may notice that we're not taking our own advice throughout this book and making our code snippets ready for translation. The reason for this is that we want to provide short and to-the-point code samples that allow for the best readability. In real-world practice, you should always internationalize your plugin.*

Why Internationalize?

Internationalizing your plugin can benefit both you and your plugin users. Compared to the more complex functionality you'll likely use when developing a plugin, internationalization will seem much easier after you follow the guidelines set out in this chapter.

- You benefit by having a larger audience of people using your plugin.
- Users benefit by using the plugin in their language.

Some plugin authors even develop relationships with translators of their plugins after working closely together to get a translation done. Forming relationships with new people is always a benefit of working on open

source projects, and by internationalizing your plugin, you open the door for more possibilities than with plugins that aren't internationalized.

NOTE *One fun thing to do is to create your own translation based on your language and region. For example, if you're from the southern part of the United States, you could create a translation of your plugin with a bit of southern slang. This can allow you to get a feel for the process that translators go through when translating plugins.*

Understanding Internationalization in Professional Work

Generally, when preparing your plugin for translation, you would do so if it is intended for public release because many of your users may run sites in languages other than your own.

Not all plugins are for use by the public. When performing custom client development, it's not always necessary to follow the steps outlined in this chapter. If your client's site is in only a single language, there might not be a need for translation. However, some clients run multilingual sites and may use a multilingual plugin that enables their content to be read in several languages. In this case, your plugin should be internationalized. You should always check with your client to see whether internationalization is a requirement.

Although it's not always necessary to internationalize text for client work, it is considered best practice to internationalize all text. This saves you from potentially having to recode it for translation later if the client changes their mind about needing translatable text, and it's always good to stick to best practices when coding.

Also, a potential benefit to learning the tools in this chapter is having an extra bullet point on your résumé for clients in need of this skill.

Getting Your Plugin Ready for Translation

The first step is to identify the text domain that your plugin will use. This is a unique identifier and denotes all text that belongs to your plugin. The text domain must match the slug of your plugin, meaning if your plugin is contained within a folder called `pdev-plugin`, the text domain will be

pdev-plugin. It's important to note that if your plugin is hosted on Wordpress.org, the text domain must match the slug of your plugin URL, for example: wordpress.org/plugins/<slug>.

WARNING *It's important to remember that a text domain should never use a variable or constant as the value. You should also always use dashes and not underscores.*

On occasion, there may be a case for defining a custom text domain that does not match your plugin's name. To do this, you can define a custom text domain in your plugin header like so:

```
/*
 * Plugin Name: PDEV Plugin
 * Author: Michael Myers
 * Text Domain: halloween-plugin
 */
```

This step is required only if you need to set a custom text domain that differs from your plugin name.

Now that you have identified the text domain for your plugin, you'll make your plugin translatable by using the `load_plugin_textdomain()` function.

```
<?php
load_plugin_textdomain( $domain, $abs_rel_path,
$plugin_rel_path );
?>
```

This function tells WordPress to load a translation file if it exists for the user's language and accepts the following parameters:

- `$domain`: A unique string that identifies text in your plugin that has been prepared for translation. You should give this the same value as the name of your plugin folder, as described earlier.
- `$abs_rel_path`: A deprecated parameter that should no longer be used. Setting this to `false` is best.
- `$plugin_rel_path`: Relative path to the translations of your plugin from the user's plugin directory (`WP_PLUGIN_DIR`).

If you were creating a plugin with a folder name of `pdev-plugin`, your code would look like this:

```
<?php
add_action( 'init', 'pdev_load_textdomain' );

function pdev_load_textdomain() {
    load_plugin_textdomain( 'pdev-plugin', false, 'pdev-
plugin/languages' );
}

?>
```

Here, the `$domain` value is `pdev-plugin` to match the plugin folder name, the `$abs_rel_path` value is `false` because it's unneeded, and the `$plugin_rel_path` value is `pdev-plugin/languages` because this is where you store translation files. You'll also notice you are using the `init` hook to register your plugin translations.

The last parameter is the directory of the plugin (`pdev-plugin`) and a subdirectory of the plugin (`languages`). It's important to create an extra folder in your plugin called `languages` to house any translations for your plugin. If you ever get more than a handful of translations, you'll want this folder because placing all those files in the top directory of your plugin can get messy.

Echoing and Returning Strings

WordPress has many useful functions for making internationalization easy. Every time you add textual content in your plugin, you should wrap it in one of the WordPress translation functions. This will identify the text string as a translatable string within your plugin.

Each one of these functions has at least one parameter that you'll use: `$domain`. This is the unique variable used in the “Getting Your Plugin Ready for Translation” section: `pdev-plugin`. The value of this variable enables WordPress to recognize it as a part of your plugin's translation files.

When viewing the core WordPress files, you'll likely notice that `$domain` is never set. WordPress uses the default, so your plugin should have a unique string to set it apart from the core.

The __() Function

The __() function works by making your text ready for translation and returning it for use in PHP. This function accepts two parameters.

- \$text: The text that is available for translation
- \$domain: The text domain defined for your plugin

In this example, you will assign the return value of __() to a PHP variable. Note that this function uses a double underscore, not a single underscore.

```
<?php
$text = __( 'WordPress is a wonderful publishing platform.',
'pdev-plugin' );
?>
```

The _e() Function

The _e() function makes your text ready for localization. It works similarly to echo in PHP by displaying text on the screen. The function accepts two parameters: \$domain and \$text. The \$text variable is the content you want translated. Now add a fun message to the site's footer using an action hook (see [Chapter 5](#), “Hooks”).

```
<?php
// Hook our message function to the footer.
add_action( 'wp_footer', 'pdev_footer_message' );

// Function that outputs a message in the footer of the
site.
function pdev_footer_message() {

    // Output the translated text.
    _e( 'This website runs on the coolest platform ever
&mdash; WordPress.',
'pdev-plugin' );

}
```

The esc_attr__() Function

esc_attr__() is the internationalization equivalent of the esc_attr() WordPress function (see [Chapter 4](#), “Security and Performance”). It escapes

HTML attributes, so anything passed to it won't break HTML validation standards or open up a site to potential security vulnerabilities.

`esc_attr__()` returns the translation for use in PHP. Now create a function that returns a link to a terms of service page on an example site and display it.

```
<?php
// A function that returns a link to the website's terms of
service page.
function pdev_terms_of_service_link() {

    return '<a href="https://example.com/tos" title="" .
        esc_attr__( 'Visit the Terms of Service page',
        'pdev-plugin' ) . '">' .
        __( 'Terms of Service', 'pdev-plugin' ) . '</a>';

}

// Display the output of the pdev_terms_of_service_link()
function.
echo pdev_terms_of_service_link();
?>
```

The `esc_attr_e()` Function

`esc_attr_e()` works the same as the `esc_attr__()` function except that it displays the translation on the screen. For example, you might display a link to the dashboard page in the WordPress admin, so you want to make sure the `title` attribute of the link works correctly. You can also use the `_e()` function from earlier.

```
<a href="<?php echo admin_url(); ?>" title="<?php esc_attr_e( 'Visit the WordPress
dashboard',
        'pdev-plugin' ); ?>">
    <?php _e( 'Dashboard', 'pdev-plugin' ); ?>
</a>
```

The `esc_html__()` Function

`esc_html__()` is the equivalent of the WordPress function `esc_html()` (see [Chapter 4](#)) for translations. You need to use this function in situations in

which HTML would be inappropriate for use. This function returns its output for use in PHP.

Suppose a form was submitted with the content of a `<textarea>` in which a default text message is provided. You'd escape the input the user submitted or escape the default message that the translator provides.

```
<?php

function pdev_get_text_message() {

    /* If the user input any text, escape it. */
    if ( !empty( $_POST['pdev-text'] ) )
        $message = esc_html( $_POST['pdev-text'] );

    /* If no text was input, use a default, translated
    message. */
    else
        $message = esc_html__( 'No message input by the
    user.', 'pdev-plugin' );

    return $message;
}
?>
```

The `esc_html_e()` Function

`esc_html_e()` behaves the same as the `esc_html__()` function except that it displays the translated text on the screen instead of returning it. For example, you may be adding a form with some default text in a `<textarea>` but want to make sure no HTML is shown.

```
<textarea name="pdev-text" id="pdev-text">
    <?php esc_html_e( 'Please input a description.', 'pdev-
    plugin' ); ?>
</textarea>
```

The `_x()` Function

Sometimes, you need to provide context for translations. The `_x()` function does the same thing as the `__()` function described earlier, except `_x()` accepts an additional parameter `$context`. This function's purpose is to provide a context in which a specific text string is used. The parameters available are as follows:

- `$text`: The text that is available for translation
- `$context`: Context information for the translators
- `$domain`: Text domain defined for your plugin

Suppose you're creating a plugin in which you use the text *Pounds* in several places. *Pounds* has many different meanings depending on the context. It could mean a unit of weight, or it could be British currency. You can provide a context for these particular instances of *Pound*.

```
<?php  
    _x( 'Pounds', 'Unit of Weight', 'pdev-plugin' );  
    _x( 'Pounds', 'British currency', 'pdev-plugin' );  
?>
```

The `_ex()` Function

`_ex()` is a function to use when you need to note a specific context for a string of text. It works the same as the `_x()` function except that it echoes its output instead of returning it.

You may use the same text in several places throughout your plugin, but each instance means something different. The term *post* is often used in blogging systems as both a noun and a verb. When internationalizing, you need to mark the difference between the two by using a context.

Use the second parameter, `$context`, to provide a context to translators on how the term is used in this instance.

The following example shows two uses of the term *post* and how it can be used as both a noun and a verb:

```
<?php  
    // Displaying "Post" as a noun.  
    _ex( 'Post', 'noun', 'pdev-plugin' );  
  
    // Displaying "Post" as a verb.  
    _ex( 'Post', 'verb', 'pdev-plugin' );  
?>
```

Well-written text is important. Before using a contextual translation function, ask yourself if the text itself can be written in a more intuitive manner. Instead of using a generic term such as *post*, you can make this easier to understand for plugin users and translators.

Post as a noun can be better written as “Select a post,” and *post* as a verb could be better written as “Post a comment.” Therefore, you wouldn’t need to use the `_ex()` function in either case. You could use `_e()` instead.

```
<?php  
    _e( 'Select a post', 'pdev-plugin' );  
    _e( 'Submit post', 'pdev-plugin' );  
?>
```

The `esc_attr_x()` Function

`esc_attr_x()` is a marriage between two of the earlier translation functions: `esc_attr__()` and `_x()`. It enables you to translate text, provide a context for translation, and escape it for use in HTML attributes. This function returns translated text for use in your plugin, but it does not have a similar function for printing text to the screen.

In the following example, the function displays a link to the WordPress admin. Use the `esc_attr_x()` function in the `title` attribute of the link so that any unwanted characters are properly escaped and to provide a context for the text string “Admin.”

```
<?php  
function pdev_plugin_display_post_link( $post_id ) {  
    // The text for the link.  
    $pdev_link_text = _x(  
        'Admin',  
        'admin link',  
        'pdev-plugin'  
    );  
  
    // The text for the “title” attribute of the link.  
    $pdev_link_title = esc_attr_x(  
        'Admin',
```

```

        'admin_link',
        'pdev-plugin'
    );

    /* Display the link on the screen. */
    echo '<a href="' . admin_url() . '"'
        . 'title="' . $pdev_link_title . '">' . $pdev_link_text .
    '</a>';
}

?>

```

The esc_html_x() Function

`esc_html_x()` merges the `esc_html__()` and `_x()` functions into a single function that allows for text translation, escapes unwanted HTML, and provides a context to translators.

Suppose you created a plugin that allows users to fill in a form about their favorite things and submit it to the site owner for review. Further suppose you have an optional input field called `pdev-favorite-food` that needs a default value translated in the case of the users not disclosing their favorite food. In this example, you use the term “None,” which is a common word and may be used in various circumstances. You should provide a context such as “favorite food” or “favorite item” to differentiate this instance of “None” from others.

```

<?php

function pdev_get_favorite_food() {

    // If the user input a favorite food.
    if ( !empty( $_POST['favorite-food'] ) )
        $pdev_favorite_food = esc_html( $_POST['favorite-
food'] );

    // If no favorite food was chosen, set a default.
    else {
        $pdev_favorite_food = esc_html_x(
            'None',
            'favorite food',
            'pdev-plugin'
        );
    }
}

```

```
        return $pdev_favorite_food;  
    }  
?>
```

The `_n()` Function

As a developer, you may not always know how many items will be returned for use in your code. You can use the `_n()` function to differentiate between the singular and plural forms of text. Not only will this function figure out which form should be used, it also will enable you to make each form translatable. Both the singular and plural forms need to be internationalized because the order in which words appear for plural and singular forms is different in various languages.

The `_n()` function's parameters are different from some of the other translation functions. The accepted parameters are as follows:

- `$single`: The text to be used if the number is singular
- `$plural`: The text to be used if the number is plural
- `$number`: The number to compare against to use either the singular or plural form
- `$domain`: Text domain defined for your plugin

`$single` represents the singular version of the text, and `$plural` represents the plural version of the text. `$number` is a parameter that you cannot know at the moment of writing your code. It's an unknown integer that can have various values.

Not all languages use only two forms (singular and plural). However, you only need to take care of these two forms. If a language requires more than two forms, translators will provide this in the translation files and WordPress' localization process will use the correct form.

Now create a function that counts the number of posts published on the site and prints the value in a sentence using the `printf()` function (see the “Using Placeholders” section).

```
<?php  
function pdev_count_published_posts() {
```

```

// Count the number of posts.
$pdev_count_posts = wp_count_posts();

// Get the count for the number of posts with a
post_status of 'publish'.
$count = $pdev_count_posts->publish;

// Display a sentence, letting the user know how many
posts are published.
printf( _n(
    'You have published %s post.', 'You have published
%s posts.',

    $count,
    'pdev-plugin' ),
    $count );
}

?>

```

The two sentences used look similar in English, and many developers may think it's easier to get away with “You have published %s post(s).” Although this works in some languages, it likely won't work in most.

For an example of why this method wouldn't work, look at the word *journal* in French. Using “journal(s)” in this case wouldn't apply because the plural form of *journal* is *journaux*.

The _nx() Function

The _nx() function is a combination of the _n() and _x() translation functions. It allows for the differentiation of singular and plural forms of text and a context for the text.

In this example, you create a function that grabs all of a site's post tags and lists the number of posts that have been given each particular tag. The _nx() function provides a way for you to display the text based on the post count of each tag and provide a context for translation of the text.

```

<?php

function pdev_list_post_tag_counts() {

    // Get all post tags in an alphabetical list.
    $tags = get_terms( 'post_tag', array( 'orderby' =>

```

```

'name', 'order' => 'ASC' ) );
// Open unordered list.
echo '<ul>';

// Loop through each post tag and display its post count
and name.
foreach ( $tags as $tag ) {
    echo '<li> ' . $tag->name . ' &mdash; ';
    printf(
        _nx(
            '%s post',
            '%s posts',
            $tag->count,
            'post count',
            'pdev-plugin'
        ),
        $tag->count
    );
    echo '</li>';
}
// Close unordered list.
echo '</ul>';
}
?>

```

The `_n_noop()` Function

There are some cases in which you might have singular and plural forms of text that you don't want translated on the spot but need translated later. This is useful when you have large lists of messages but don't know which to display until a variable has been set.

The `_n_noop()` function adds these values to the translation files. Rather than returning a translated string like most other translation functions, it returns an array with both values.

Suppose you created two custom post types (see [Chapter 8](#), “Content”) called `video` and `music` to give your plugin users some nifty features for their site. You have some messages you'd like translated, but you want to keep the code easy to reuse and short. In the following example, you create a function that takes a parameter of `$post_type`. This displays the appropriate message depending on the value of this parameter.

```

<?php

function pdev_count_posts_of_custom_types( $post_type =
'video' ) {

    // Get a count of all posts of the given post type.
    $all_posts = wp_count_posts( $post_type );

    // Get the count of the published posts.
    $count = $all_posts->publish;

    // Prepare an array of messages.
    $pdev_messages = array(
        'video' => __n_noop( 'You have %s video.', 'You have
%s videos.',
        'pdev-plugin' ),
        'music' => __n_noop( 'You have %s music file.', 'You have %s music files.', 'pdev-plugin' )
    );

    // Get the message for the custom post type given.
    $pdev_message = $pdev_messages[$post_type];

    // Print the message for the custom post type given and
    // its count.
    printf( __n(
        $pdev_message['singular'],
        $pdev_message['plural'],
        $count,
        'pdev-plugin'
    ), $count );

}

?>

```

The previous example uses the `__n_noop()` function to build the array of messages, but it used the `__n()` function to display the translated message. This enabled you to translate only the message needed at the moment instead of each message.

The `__nx_noop()` Function

`__nx_noop()` combines the `__n_noop()` function and the `_x()` function to enable setting up text for later translation and providing a context for translators on how the text is used in the plugin. It works the same as

`_n_noop()` by adding the text to translation files but not translating it when used in PHP.

Building off the previous example of showing the number of posts published by type, you can use the `_nx_noop()` function to add a context.

```
<?php

function pdev_count_posts_of_custom_types( $post_type =
'video' ) {

    // Get a count of all posts of the given post type.
    $all_posts = wp_count_posts( $post_type );

    // Get the count of the published posts.
    $count = $all_posts->publish;

    // Prepare an array of messages.
    $pdev_messages = array(
        'video' => _n_noop(
            '%s video',
            '%s videos',
            'video post count',
            'pdev-plugin'
        ),
        'music' => _n_noop(
            '%s music file',
            '%s music files',
            'music post count',
            'pdev-plugin'
        )
    );

    // Get the message for the custom post type given.
    $pdev_message = $pdev_messages[$post_type];

    // Print the message for the custom post type given and
    // its count.
    printf( _n(
        $pdev_message['singular'],
        $pdev_message['plural'],
        $count,
        'pdev-plugin'
    ), $count );

}

?>
```

Using Placeholders

You may have noticed the use of symbols such as %s and %1\$s in previous examples. These are placeholders for variables. Placeholders are useful because they enable you to translate strings without breaking them apart.

The translation functions in WordPress cannot output placeholders on their own. The placeholders are merely there for translators to properly set within the text of their translation files. Placeholders must be converted to a given variable in PHP.

The `printf()` and `sprintf()` PHP functions are useful when using placeholders. Both functions can replace the placeholder with a given variable. Use `printf()` to print text to the screen and `sprintf()` to return text. Each function takes in a first parameter of `$text`, which is the text you're translating. Both can then receive any number of extra parameters that represent the placeholders in the `$text` variable.

Now take a look at the following example of a translated sentence that works in English but breaks in many other languages:

```
<?php
function pdev_display_blog_name() {

    _e( 'The name of your blog is ', 'pdev-plugin' );
    echo get_bloginfo( 'name' );
    _e( '.', 'pdev-plugin' );

}?>
```

Although the text in that function is internationalized, it's not done in a way that's easily translatable. This is where placeholders come in. They enable you to set a variable in the text and keep it as one sentence.

Now rewrite that function in a way that makes it easier for translators to translate. Use `printf()` to print the sentence to the screen and convert the placeholders.

```
<?php
function pdev_display_blog_name2() {

    printf(
        __( 'The name of your blog is %s.', 'pdev-plugin' ),
        get_bloginfo( 'name' )
    )
}
```

```
 );
}

?>
```

Now create a function that returns the tagline of the site in a sentence using the `sprintf()` function.

```
<?php
function pdev_get_blog_tagline() {

    return sprintf(
        __( 'The tagline of your site is %s.', 'pdev-plugin'
    ),
        get_bloginfo( 'description' )
    );

}
?>
```

Sometimes you need multiple placeholders in one text string. Luckily, both `printf()` and `sprintf()` handle this wonderfully. The big difference here is that you shouldn't use `%s`. It's best to use numbered placeholders instead because the order of words in other languages may be different than your own.

In the following example, you use multiple placeholders to display a sentence depending on the number of posts published on the blog, along with the blog title.

```
<?php

function pdev_display_blog_name_and_post_count() {

    // Get the number of posts.
    $count_posts = wp_count_posts();

    // Get the number of published posts.
    $count = $count_posts->publish;

    // Get the site name.
    $site_name = get_bloginfo( 'name' );

    // Display a sentence based on the number of posts
    // published.
    printf(
```

```

        _n(
            'There is %1$s post published on %2$s.',
            'There are %1$s posts published on %2$s.',
            $count,
            'pdev-plugin'
        ),
        $count, $site_name
    );
}

?>

```

In the example function, the %1\$s placeholder represents the \$count variable, which returns the number of published posts. The %2\$s placeholder represents the value of \$site_name, which was set to the name of the site.

Internationalizing JavaScript

Some plugins require JavaScript to function properly (see [Chapter 6](#), “JavaScript”). Because the internationalization functions in WordPress are written in PHP, you can't use them inside JavaScript files. This makes it a little tougher to translate but not impossible.

WordPress provides a function called `wp_localize_script()` that enables you to pass translated text to an external file. You can then use the translated strings within your JavaScript. Take a look at what this function looks like:

```

<?php
wp_localize_script( $handle, $object_name, $l10n );
?>

```

- \$handle must match the \$handle parameter of a script that's already registered with WordPress (see [Chapter 6](#)).
- \$object_name is a unique identifier that represents this set of translations.
- \$l10n is an array of translations with named keys that each has a value of a single translated string.

To understand how all this comes together, you need to create a plugin that uses JavaScript. You use a simple script here. To delve more into the process of using JavaScript, read the thorough explanation in [Chapter 6](#). Your plugin will add two input buttons to the site's footer. When either of the buttons is clicked, a translated message appears.

The first step is to create a new plugin folder called pdev-alert-box and place a new PHP file called pdev-alert-box.php in this folder. In the pdev-alert-box.php file, add your plugin information ([Chapter 2](#), “Plugin Framework”).

```
<?php
/**
 * Plugin Name: PDEV Alert Box
 * Plugin URI: https://example.com
 * Description: A plugin example that places two input
buttons
    in the blog footer that when clicked display an alert
box.
 * Version: 0.1
 * Author: WROX
 * Author URI: https://wrox.com
 */
```

Next, you need to load your translation as described in the “Getting Your Plugin Ready for Translation” section.

```
/* Add the translation function after the plugins loaded
hook. */
add_action( 'plugins_loaded',
'pdev_alert_box_load_translation' );

/**
 * Loads a translation file if the page being viewed isn't
in the admin.
 *
 * @since 0.1
 */
function pdev_alert_box_load_translation() {

    /* If we're not in the admin, load any translation of
our plugin. */
    if ( !is_admin() )
        load_plugin_textdomain( 'pdev-alert-box', false,
```

```
'pdev-alert-box/languages' );
}
```

At this point, you need to load your script using the `wp_enqueue_script()` function. After calling that function, you can localize your script using the `wp_localize_script()` function. It is important that this function is called after the script has been registered because the `$handle` variable has to be set.

```
/* Add our script function to the print scripts action. */
add_action( 'wp_print_scripts', 'pdev_alert_box_load_script'
);

/**
 * Loads the alert box script and localizes text strings
 * that need translation.
 *
 * @since 0.1
 */
function pdev_alert_box_load_script() {

    /* If we're in the WordPress admin, don't go any
     * farther. */
    if ( is_admin() )
        return;

    /* Get script path and file name. */
    $script = trailingslashit( plugins_url( 'pdev-alert-box'
) )
        . 'pdev-alert-box-script.js';

    /* Enqueue our script for use. */
    wp_enqueue_script( 'pdev-alert-box', $script, false, 0.1
);

    /* Localize text strings used in the JavaScript file. */
    wp_localize_script( 'pdev-alert-box',
'pdev_alert_box_L10n', array(
    'pdev_box_1' => __( 'Alert boxes are annoying!', 'pdev-alert-box' ),
    'pdev_box_2' => __( 'They are really annoying!', 'pdev-alert-box' ),
) );
}
```

Now you add a couple of fun input buttons to the footer of the site. Notice the use of the `esc_attr__()` function from earlier in the chapter to translate and escape the `value` attributes of the buttons.

```
/* Add our alert box buttons to the site footer. */
add_action( 'wp_footer', 'pdev_alert_box_display_buttons' );

/**
 * Displays two input buttons with a paragraph. Each button
has an onClick()
 * event that loads a JavaScript alert box.
 *
 * @since 0.1
 */
function pdev_alert_box_display_buttons() {

    /* Get the HTML for the first input button. */
    $pdev_alert_box_buttons = '<input type="button"
onclick="pdev_show_alert_box_1()"
    value="' . esc_attr__( 'Press me!', 'pdev-alert-box' ) .
'"/>';

    /* Get the HTML for the second input button. */
    $pdev_alert_box_buttons .= '<input type="button"
onclick="pdev_show_alert_box_2()"
    value="' . esc_attr__( 'Now press me!', 'pdev-alert-box' )
    . '"/>';

    /* Wrap the buttons in a paragraph tag. */
    echo '<p>' . $pdev_alert_box_buttons . '</p>';
}
?>
```

Your plugin's PHP file is complete. You now need to add a JavaScript file called `pdev-alert-box-script.js` to your plugin folder. After it's created, you can add two functions for displaying the alert boxes on the screen. Within the `pdev-alert-box-script.js` file, add the JavaScript.

```
/**
 * Displays an alert box with our first translated message
when called.
*/
function pdev_show_alert_box_1() {
    alert( pdev_alert_box_L10n(pdev_box_1) );
}
```

```
/***
 * Displays an alert box with our second translated message
when called.
*/
function pdev_show_alert_box_2() {
    alert( pdev_alert_box_L10n.pdev_box_2 );
}
```

Because you're working across multiple files, it may be hard to see how the files interact with one another.

The two most important parameters from the call to `wp_localize_script()` are the `$object_name` and `$l10n` parameters. In the example plugin, you specifically set `$object_name` to `pdev_alert_box_L10n` and `$l10n` to an array of key/value pairs.

When needing a translation in the JavaScript file, you used `$object_name.$l10n[$key]`. In the JavaScript function for the first alert box, this looks like this:

```
alert( pdev_alert_box_L10n.pdev_box_2 );
```

Developer Handbook Resource

This chapter covers a lot of information about proper ways to internationalize your plugin to support languages across the globe. As always, the [WordPress.org Plugin Developer Handbook](https://developer.wordpress.org/plugins/internationalization/how-to-internationalize-your-plugin) is a great resource for digging into each one of these topics at a deeper level.

<https://developer.wordpress.org/plugins/internationalization/how-to-internationalize-your-plugin>

CREATING TRANSLATION FILES

Now that you've done all the hard work of internationalizing the text strings in your plugin, you have one more step to complete. This step requires much less work than the previous steps. You need to create a default translation file to kick-start the translation process for potential translators.

To provide a consistent look at how this process works, you can work with the PDEV Alert Box plugin from the “Internationalizing JavaScript” section.

The MO and PO Files

When translators create translations of your plugin, they use your plugin's portable object template (POT) file to create two files: `pdev-alert-box-$locale.mo` and `pdev-alert-box-$locale.po`. The portable object (PO) files are translation files that are then translated to machine object (MO) files. The MO files are loaded by the user's WordPress installation to provide the translated strings to the plugin.

`pdev-alert-box` is the `$domain` parameter used in the translation functions throughout the plugin. `$locale` is a variable that represents the language and regional dialect. For example, WordPress uses `en_US` as its default language. `en` represents English, and `US` represents the region.

The PO file is the file used by translation tools to allow for human-readable text. Translators work with this file to provide a translation. The PO file isn't necessary to run a translation of your plugin, but it's always nice to package it with your plugin download for use by other users who may want to update any text to suit their needs.

The MO file is created from the finished translation. WordPress uses it to translate internationalized text strings from your plugin.

WordPress users set their locale in their `wp-config.php` file. This tells WordPress to look for any translations with that locale and load them. If users set their locale to `fr_FR` (French), WordPress would load a file called `pdev-alert-box-fr_FR.mo` if it existed in your plugin.

Translation Tools

Many translation tools around the web are open source and free to download. Each tool isn't covered in detail here because they all have different ways to create translations. However, there is a list of supported translation tools for WordPress.

The following are available tools for translation:

- **Poedit:** <https://poedit.net>
- **GlotPress:** <https://glotpress.blog>
- **WP CLI:** <https://developer.wordpress.org/cli/commands/i18n>

- **Launchpad:** <https://translations.launchpad.net>
- **KBabel:** <https://i18n.kde.org>
- **GNU Gettext:** <https://gnu.org/software/gettext>

One of the most common tools for plugin developers is Poedit. It has a simple, point-and-click interface that enables developers to create a POT file from their plugin.

GlotPress is a new web-based tool from the people behind the WordPress project that promises to enable a single person or a team to work on translating software. It is now being used to facilitate the translation process for the WordPress software at <https://translate.wordpress.org>.

How to Create a POT File

Using Poedit from the “Translating Tools” section, you can create a POT file. You need to input only a few pieces of information, and Poedit does the rest for you.

1. Select File \Rightarrow New on the main menu.
2. Set the language of the translation. In this example, you'll use English.
3. Before you can add sources, you need to save the file.
4. Save your file as `plugin-name.pot` in your plugin's `languages` folder.
For example, the PDEV Alert Box plugin would be `pdev-alert-box.pot`.
5. Select the Extract From Sources button.
6. A Settings box appears with three tabs, as shown in [Figure 11-1](#).

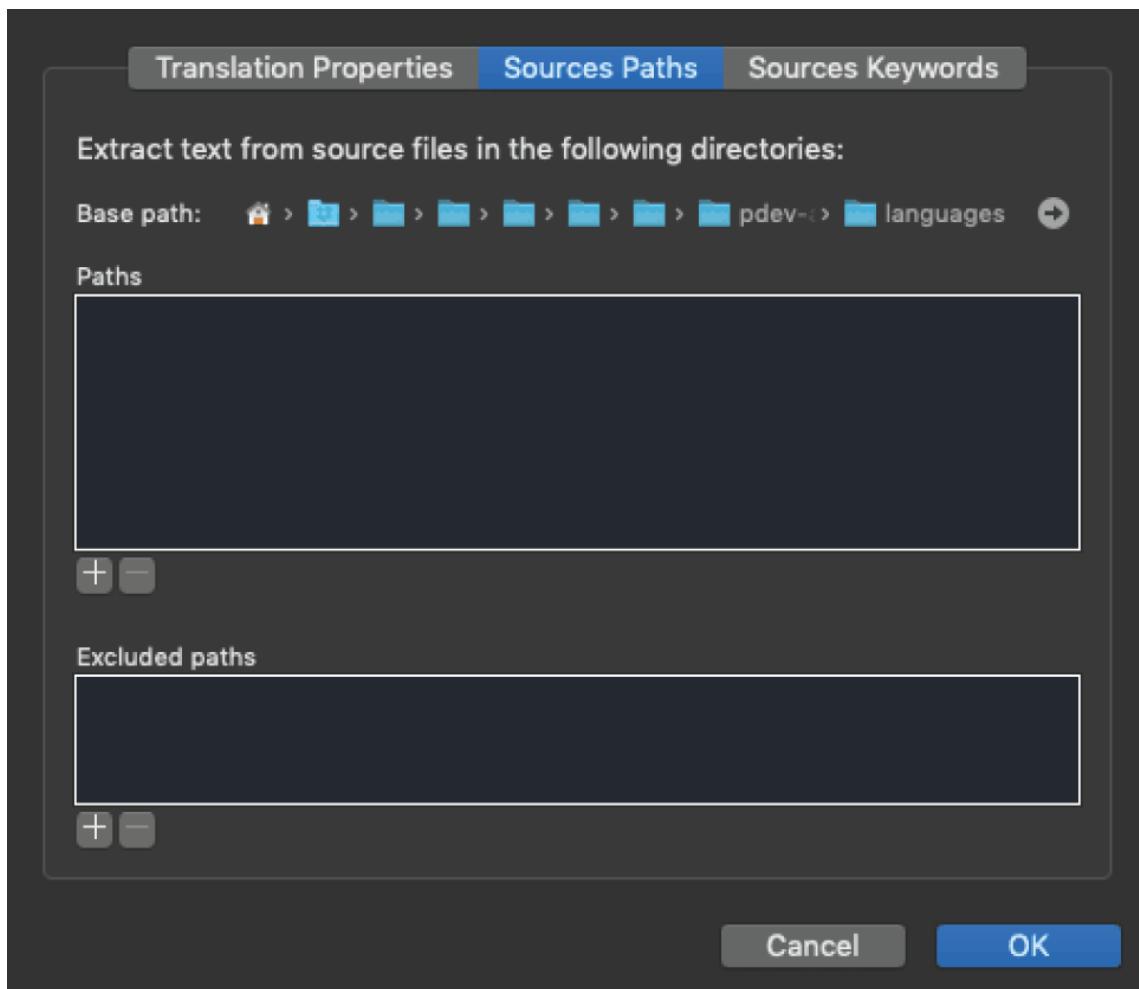


FIGURE 11-1: Settings box

7. On the Translation Properties tab, fill in the input boxes that are relevant to your plugin, leaving the Charset box as UTF-8.
8. On the Sources Paths tab, add a new path to the root directory for your plugin.
9. On the Sources Keywords tab, enter each function name from the “Echoing and Returning Strings” section of this chapter as a new keyword. For example, enter `esc_attr_e()` as `esc_attr_e`.
10. Click OK and save your POT file settings.

After you complete this process, Poedit synchronizes your POT file with your plugin, and you will have completed the process of preparing your plugin for translation.

Command Line

Another popular option for generating a POT file is using the WP CLI i18n commands. Simply run the following WP CLI command to create a POT file for your WordPress plugin in the current directory:

```
# Create a POT file for the WordPress plugin/theme in the
# current directory
$ wp i18n make-pot . languages/my-plugin.pot
```

For more information, visit the official WP CLI documentation website:

<https://developer.wordpress.org/cli/commands/i18n>

Where to Store Translation Files

Many plugins add translation files in the top level of their plugin folder. Although translations will work when using this method, it can get a bit messy and is discouraged. If you have a plugin with many other files, it may become much too unorganized if you take this route.

For the cleanest, most organized system, create an extra directory in your plugin folder called `languages`. When you release your plugin, you can store your default translation files in this folder. When translators send you translation files for your plugin, you simply drop those files in the `languages` folder. Be sure to add both the PO file (for translators) and the MO file (for WordPress) of the translation you're given.

SUMMARY

The biggest lesson to take away from this chapter is that people use WordPress in all corners of the world. To have a good plugin for public download, it is essential that you prepare it for translation. This can set your plugin apart from the many thousands of plugins that aren't internationalized. After you start putting the tools of this chapter into practice, you can see how easy it is to make your plugin more powerful and build a larger community around your work.

12

REST API

WHAT'S IN THIS CHAPTER?

- Introduction to the REST API
- Routes and endpoints
- REST API clients
- Basic and enhanced authentication
- Custom endpoints
- The HTTP API
- Requests and responses
- Bringing everything together

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are found at www.wiley.com/go/prowordpressdev2e on the Downloads tab.

It was June 2013 when the WordPress REST API project was first born as a plugin, and by 2015 it had been merged into WordPress Core, making it available to everyone without the need of a separate plugin. Since then, this API has been iterated on and improved, helping it become one of the most important things to understand for professional WordPress plugin development.

WHAT THE REST API IS

Representational State Transfer (REST) is a set of constraints that are useful for creating web services. These six fundamental constraints are as follows:

- **Uniform interface:** The URLs used for access in the system need to be uniform, consistent, and accessible using a common approach such as GET.
- **Client-server:** Clients and servers may evolve separately without any dependency on each other as long as the interface between them is not altered.
- **Stateless:** The server does not store anything about the latest HTTP request made by the client. Every request is treated as a new request.
- **Cacheable:** All resources must be cacheable. Caching can be implemented on the server or client side.
- **Layered system:** This system allows for a layered system architecture where a client cannot tell whether it is connected to an intermediary server or the end server.
- **Code on demand (optional):** Most of the time you will be sending static representations of resources in the form of XML or JSON. This optional constraint allows you to return executable code, such as JavaScript, to support part of your application.

Together, these constraints provide a holistic approach to building a complete and scalable web application with proper separation of duties, while also being relatively simple to interface with.

In a RESTful web application, requests are sent to specific URIs, and depending on the type of request, the URI, and the payload sent with it, the web application will send back any number of different responses, usually formatted in JSON, HTML, or XML.

In WordPress, this boils down to sending various types of HTTP requests to specific URIs (commonly referred to as *endpoints*) to retrieve and/or manipulate some data (likely from a cache or the database) in a secure and performant way.

WHAT YOU CAN DO WITH THE REST API

Practically speaking, the WordPress REST API provides a nonvisual window into creating, reading, updating, and deleting (CRUD) anything

that happens to be stored inside WordPress. You'll use the following HTTP methods when interacting with WordPress via the REST API:

- **GET**: Retrieves data from the server such as a post
- **POST**: Adds data to the server, such as a post, comment, or even media attachments
- **PUT**: Used to edit or update existing data on the server, such as a post
- **DELETE**: Removes data from the server, like deleting a user

Where you would normally click around through a website or the admin area, now you can access all the same functionality without the user interface, instead using a series of requests to make your own interfaces using the data you've requested. In short, you can build really cool applications that do not necessarily look, act, or feel like a web application, never revealing to anyone that it is using WordPress to power it all. This truly transforms WordPress into an application framework.

ACCESSING THE WORDPRESS REST API

All newer versions of WordPress have the REST API available by default. You can perform a simple **GET** request to retrieve data directly via your browser by visiting <https://example.com/wp-json/wp/v2>.

This request will return a JSON response with all sorts of information about the WordPress REST API. Adding specific endpoints to the URL will allow you to retrieve specific types of data from WordPress. As an example, let's retrieve a list of users.

```
https://example.com/wp-json/wp/v2/users
```

As you can see, this returns a list of users within your WordPress website who have published a piece of content. Another useful example is using a simple argument to search for a subset of the data returned. For example, let's search all published posts that match the search keyword *Halloween*.

```
https://example.com/wp-json/wp/v2/posts?search=halloween
```

These are basic examples that show how to access the WordPress REST API in browser. Throughout this chapter, you'll cover many different ways of retrieving and working with data within WordPress.

When working with JSON data, it can be helpful to install a browser extension to format the data in a more readable manner. If you are using Chrome, I recommend the JSON Formatter extension located at <http://bit.ly/2QQL4wy>. You can see how unformatted JSON looks in [Figure 12-1](#) compared to the formatted JSON in [Figure 12-2](#).

```
[{"id":1,"name":"Brad","url":"","description":"","link":"http://localhost/prowp/blo
g\author\brad\/","slug":"brad","avatar_urls":
{"24":"http://2.gravatar.com/avatar\/e8e42af2e29adbc63b2f45e57ed74b48?
s=24&d=mm&r=g","48":"http://2.gravatar.com/avatar\/e8e42af2e29adbc63b2f45e57ed74b48?
s=48&d=mm&r=g","96":"http://2.gravatar.com/avatar\/e8e42af2e29adbc63b2f45e57ed74b48?
s=96&d=mm&r=g"},"meta":[],"_links": {"self": [{"href": "http://localhost/prowp/wp-
json/wp/v2/users/1"}]}, "collection": [{"href": "http://localhost/prowp/wp-
json/wp/v2/users"}]}},
 {"id":3,"name":"jasonvoorhees","url":"","description":"","link":"http://localhost/pr
owp/blog\author\jasonvoorhees\/","slug":"jasonvoorhees","avatar_urls":
 {"24":"http://2.gravatar.com/avatar\/eba69e62f8bc92297b7a97659b5d6130?
s=24&d=mm&r=g","48":"http://2.gravatar.com/avatar\/eba69e62f8bc92297b7a97659b5d6130?
s=48&d=mm&r=g","96":"http://2.gravatar.com/avatar\/eba69e62f8bc92297b7a97659b5d6130?
s=96&d=mm&r=g"},"meta":[],"_links": {"self": [{"href": "http://localhost/prowp/wp-
json/wp/v2/users/3"}]}, "collection": [{"href": "http://localhost/prowp/wp-
json/wp/v2/users"}]}},
 {"id":2,"name":"michaelmyers","url":"","description":"","link":"http://localhost/pro
wp/blog\author\michaelmyers\/","slug":"michaelmyers","avatar_urls":
 {"24":"http://0.gravatar.com/avatar\cf196788a87db6842a0096c9883d0c5d?
s=24&d=mm&r=g","48":"http://0.gravatar.com/avatar\cf196788a87db6842a0096c9883d0c5d?
s=48&d=mm&r=g","96":"http://0.gravatar.com/avatar\cf196788a87db6842a0096c9883d0c5d?
s=96&d=mm&r=g"},"meta":[],"_links": {"self": [{"href": "http://localhost/prowp/wp-
json/wp/v2/users/2"}]}, "collection": [{"href": "http://localhost/prowp/wp-
json/wp/v2/users"}]}]
```

[FIGURE 12-1](#): Unformatted JSON

```
▼ [
  ▼ {
    "id": 1,
    "name": "Brad",
    "url": "",
    "description": "",
    "link": "http://localhost/prowp/blog/author/brad/",
    "slug": "brad",
    ▶ "avatar_urls": { ... }, // 3 items
    "meta": [],
    ▶ "_links": { ... } // 2 items
  },
  ▼ {
    "id": 3,
    "name": "jasonvoorhees",
    "url": "",
    "description": "",
    "link": "http://localhost/prowp/blog/author/jasonvoorhees/",
    "slug": "jasonvoorhees",
    ▶ "avatar_urls": { ... }, // 3 items
    "meta": [],
    ▶ "_links": { ... } // 2 items
  },
  ▼ {
    "id": 2,
    "name": "michaelmyers",
    "url": "",
    "description": "",
    "link": "http://localhost/prowp/blog/author/michaelmyers/",
    "slug": "michaelmyers",
    ▶ "avatar_urls": { ... }, // 3 items
    "meta": [],
    ▶ "_links": { ... } // 2 items
  }
]
```

FIGURE 12-2: Formatted JSON

Default Endpoints

In the previous section, you hit two different REST API endpoints: users and posts. WordPress, by default, has several endpoints available for retrieving all types of data.

- **Posts:** /wp/v2/posts/
- **Post Revisions:** /wp/v2/posts/<id>/revisions/
- **Pages:** /wp/v2/pages/
- **Page Revisions:** /wp/v2/pages/<id>/revisions/
- **Categories:** /wp/v2/categories/
- **Tags:** /wp/v2/tags/
- **Comments:** /wp/v2/comments/
- **Taxonomies:** /wp/v2/taxonomies/
- **Media:** /wp/v2/media/
- **Users:** /wp/v2/users/
- **Post Types:** /wp/v2/types/
- **Post Statuses:** /wp/v2/statuses/
- **Settings:** /wp/v2/settings/
- **Themes:** /wp/v2/themes/
- **Search:** /wp/v2/search/
- **Blocks:** /wp/v2/blocks/
- **Block Revisions:** /wp/v2/blocks/<id>/autosaves/
- **Block Renderer:** /wp/v2/block-renderer/

It's easy to see how much data is available by default from the WordPress REST API.

REST API Clients

When working with an API, whether it be the WordPress REST API or another source, it can be helpful to use a REST API client to verify the request calls and data results quickly. This can help you quickly confirm the API endpoints, that authentication is working as needed, and that the data returned is what you expect. The following sections discuss two popular REST API clients.

Insomnia

A free and open source client for Mac, Windows, and Linux, Insomnia is a great option when working with APIs. Specify the request URL, payload, headers, and authorization all in one place. To view a full list of features and to download Insomnia, visit <https://insomnia.rest>.

Postman

Postman is a popular collaboration platform for API development. Currently, Postman has apps available for Mac, Windows, or Linux systems. A proprietary option, with premium features, Postman has a good free version that supports the basics that you'll need to validate, authenticate, and test against an API. To view all features and download it, visit <https://www.getpostman.com>.

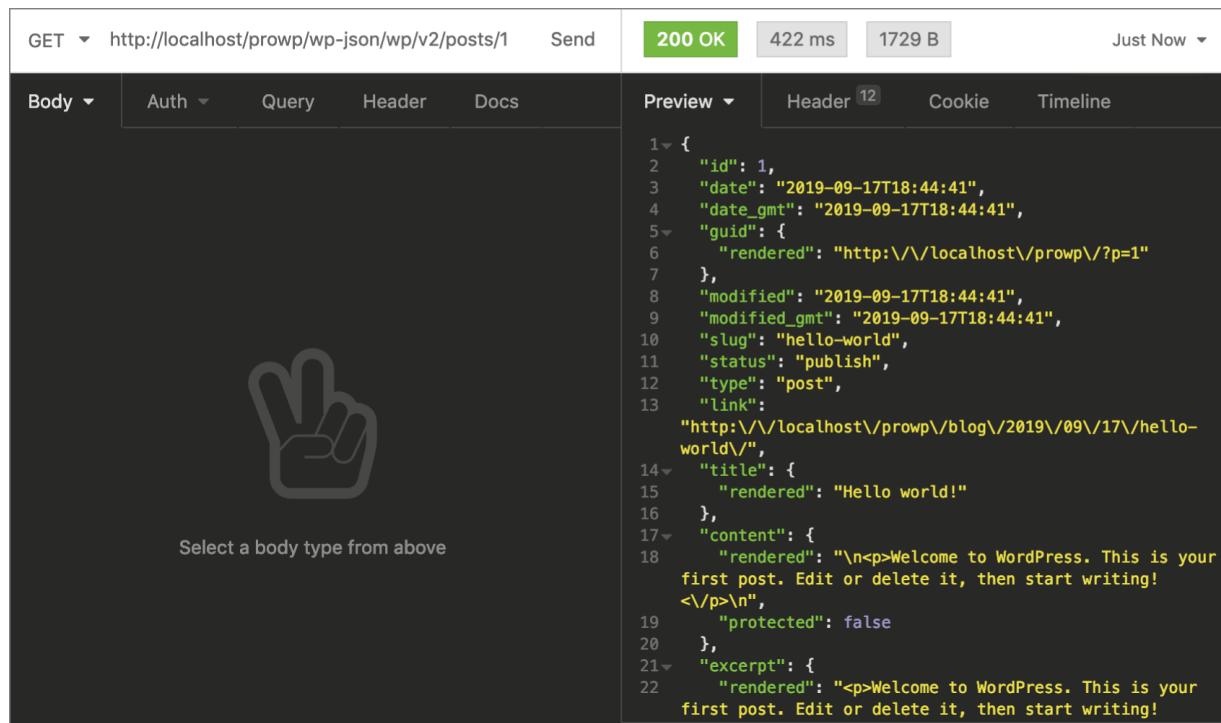
Both Insomnia and Postman give you an intuitive UI to interact with any available API. As an example, let's use a simple GET request to return post ID 1, which is most likely the Hello World example post created when you installed WordPress. First open the Insomnia app on your computer. If this is your first time running Insomnia, you will be prompted to either create a new request or import one from a file, so click New Request. Enter a name for your new request environment like **Localhost Testing** and click the Create button. Now enter the WordPress REST API request URL and click Send. In this example, the request URL will be as follows:

<http://localhost/prowp/wp-json/wp/v2/posts/1>

The results returned for post ID 1 are displayed in the Preview tab, as shown in [Figure 12-3](#).

As you can see, using a REST API client tool can quickly help you validate the data you are working with. Once you have perfected your API requests, you can begin integrating those requests into your custom WordPress plugins.

Try using a REST API client to request data from the default WordPress API endpoints you reviewed in the “Default Endpoints” section earlier in this chapter.



```

1  {
2    "id": 1,
3    "date": "2019-09-17T18:44:41",
4    "date_gmt": "2019-09-17T18:44:41",
5    "guid": {
6      "rendered": "http://localhost/prowp/?p=1"
7    },
8    "modified": "2019-09-17T18:44:41",
9    "modified_gmt": "2019-09-17T18:44:41",
10   "slug": "hello-world",
11   "status": "publish",
12   "type": "post",
13   "link": "http://localhost/prowp/blog/2019/09/17/hello-world/",
14   "title": {
15     "rendered": "Hello world!"
16   },
17   "content": {
18     "rendered": "\n<p>Welcome to WordPress. This is your first post. Edit or delete it, then start writing!\n</p>\n",
19     "protected": false
20   },
21   "excerpt": {
22     "rendered": "<p>Welcome to WordPress. This is your first post. Edit or delete it, then start writing!"

```

FIGURE 12-3: Post results

Authentication

All data that is public in WordPress can be accessed via the REST API and does not require authentication. For example, a published post on your website is public and therefore available to retrieve using the REST API. Draft posts are not public, because they are not published yet, so you will need to authenticate with WordPress to retrieve that information.

Let's look at an example accessing draft post data from the command line using SSH. If you are using macOS or Linux, you can use SSH from the terminal without any special software. On Windows, you'll need to install an SSH client like Putty (<https://www.putty.org>).

On a Mac, open your Terminal application and enter the following command, replacing `localhost` with the WordPress website you want to access:

```
curl -X GET http://localhost/wp-json/wp/v2/posts
```

This command will return a set of published posts from your local website. Now let's try retrieving draft posts.

```
http://localhost/wp-json/wp/v2/posts?status=draft
```

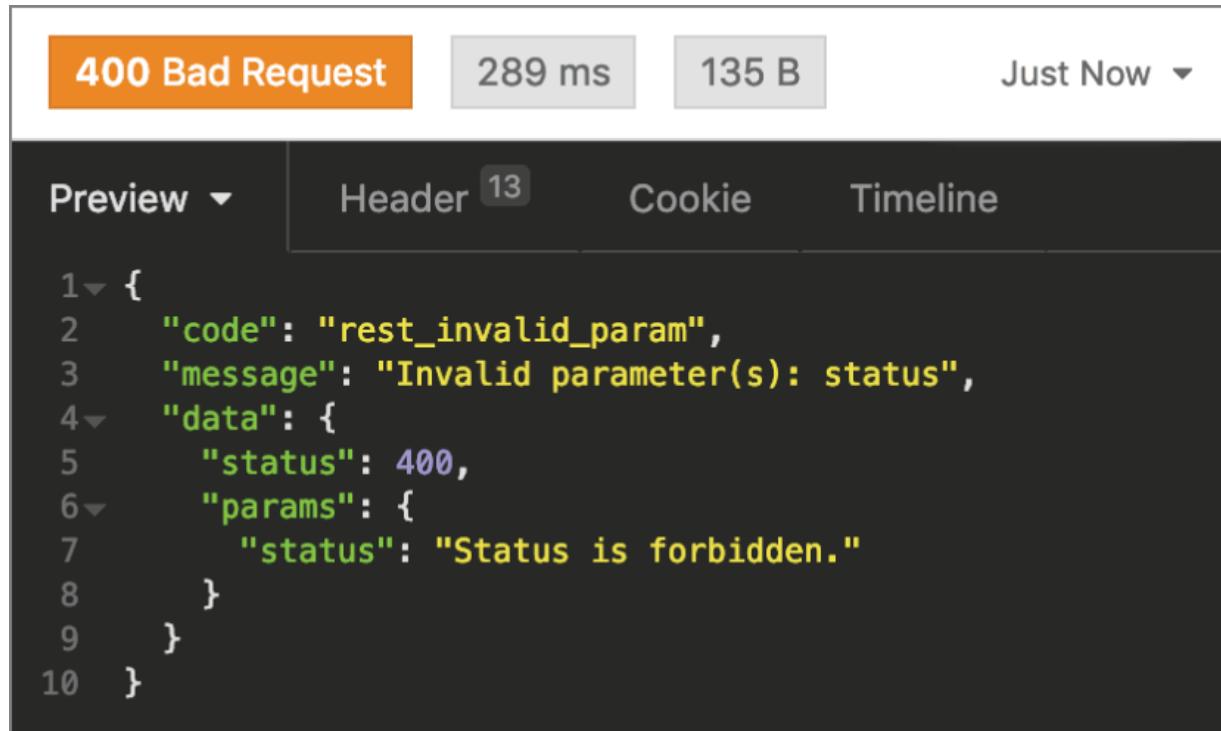
You'll notice that an error message is returned: `Status is forbidden`. This is because you have not authenticated with WordPress, so draft posts are not available. As a basic example of authentication, you'll use the [Basic Authentication handler plugin for WordPress](https://github.com/WP-API/Basic-Auth): [`https://github.com/WP-API/Basic-Auth`](https://github.com/WP-API/Basic-Auth). This plugin does exactly what it says: it uses basic authentication to send your username and password with every request. This is a basic form of authentication and should only be used for testing, as there are other, more secure methods available for production websites.

Once you have installed and activated the plugin, you can now pass your credentials through the request to access draft blog posts like so:

```
curl -X GET --user username:password -i http://localhost/wp-json/wp/v2/posts?status=draft
```

Simply replace `username` and `password` with your WordPress username and password to properly authenticate your account within the request. WordPress will now return your draft posts.

Now let's look at the same example using the Insomnia REST API client you reviewed in the previous section. Using a `GET` request without authentication, you will receive a `Status is forbidden 400` error message, shown in [Figure 12-4](#).



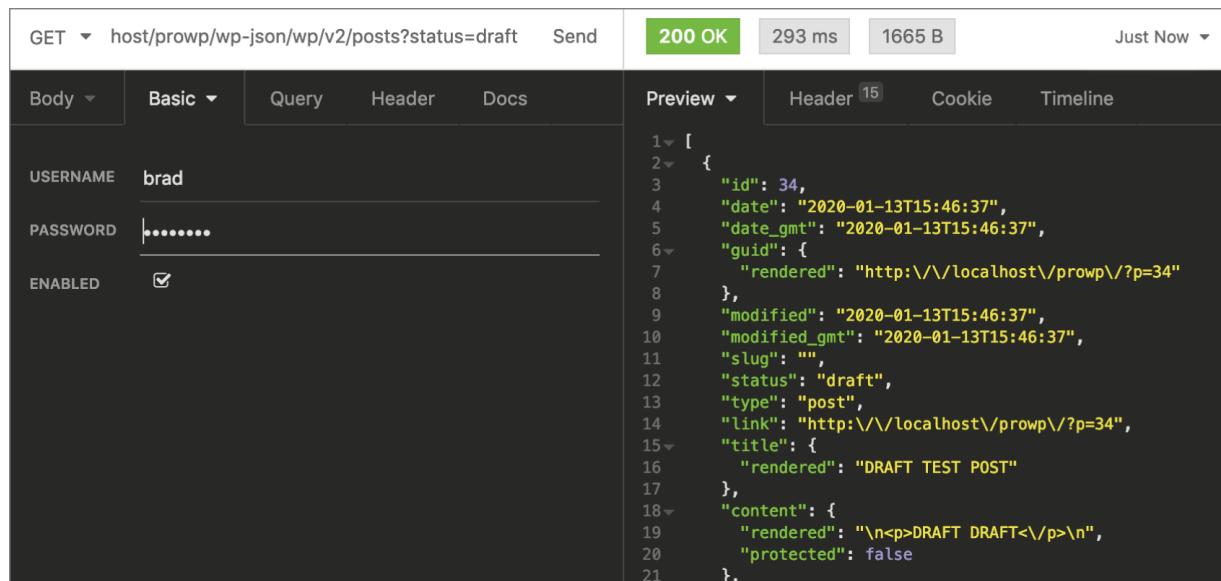
```

1  {
2      "code": "rest_invalid_param",
3      "message": "Invalid parameter(s): status",
4      "data": {
5          "status": 400,
6          "params": {
7              "status": "Status is forbidden."
8          }
9      }
10 }

```

FIGURE 12-4: Error message

To authenticate, simply click the Auth tab and select Basic Auth. Enter your username and password and send the request again. This time the request is authenticated, and the results are returned and displayed, as shown in in [Figure 12-5](#).



```

1  [
2  {
3      "id": 34,
4      "date": "2020-01-13T15:46:37",
5      "date_gmt": "2020-01-13T15:46:37",
6      "guid": {
7          "rendered": "http://localhost/prowp/?p=34"
8      },
9      "modified": "2020-01-13T15:46:37",
10     "modified_gmt": "2020-01-13T15:46:37",
11     "slug": "",
12     "status": "draft",
13     "type": "post",
14     "link": "http://localhost/prowp/?p=34",
15     "title": {
16         "rendered": "DRAFT TEST POST"
17     },
18     "content": {
19         "rendered": "\n<p>DRAFT DRAFT</p>\n",
20         "protected": false
21     }
}

```

FIGURE 12-5: Authentication worked

As you can see, this is a basic example demonstrating how to authenticate with WordPress to retrieve nonpublic data using the WordPress REST API. Later in this chapter we'll cover accessing REST API data within a custom plugin utilizing the HTTP API.

Enhanced Authentication

As shown in the previous example, basic authentication allows you to access data not publicly available from the WordPress REST API. The downside to this method is that the auth credentials are sent in plain text with every request, so as you would expect, this is not very secure. Let's look at some enhanced authentication options available when accessing the REST API.

- **Application Passwords:** Authenticate users by generating a unique password for each application without revealing the user's main account password. Application passwords can be revoked for each application individually.
 - <https://wordpress.org/plugins/application-passwords>
- **JWT (JSON Web Tokens) Authentication:** JSON Web Tokens are an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between two systems as a JSON object.
 - <https://wordpress.org/plugins/jwt-authentication-for-wp-rest-api>
- **OAuth 1.0a Server:** Use the OAuth 1.0a protocol to allow delegated authorization allowing applications to access the REST API using a set of secondary credentials.
 - <https://wordpress.org/plugins/rest-api-oauth1>
- **REST API Authentication:** A more robust option is the REST API Authentication plugin from miniOrange. This plugin supports multiple authentication options including basic auth, key authentication, JWT authentication, and OAuth 2.0. The plugin is freely available on WordPress.org. A premium version, with enhanced features and professional support, is also available.

► <https://wordpress.org/plugins/wp-rest-api-authentication>

Custom Endpoints

As with almost everything in WordPress, the REST API can be extended, making it easy to return any custom data you might need. To do this, you will register a custom endpoint for the REST API. These custom endpoints can return any type of dataset you require.

Let's create a custom endpoint that returns the first post title for a specific author in WordPress. First let's construct a custom function to return the exact data you are looking for.

```
/**  
 * Grab latest post title by the author ID  
 *  
 * @param array $data Options for the function.  
 * @return string|null Post title for the latest, * or null  
 if none.  
 */  
function pdev_return_post_title_by_author_id( $data ) {  
  
    $posts = get_posts( array(  
        'author' => absint( $data['id'] ),  
    ) );  
  
    if ( empty( $posts ) ) {  
        return new WP_Error( 'no_author', 'Invalid author',  
array( 'status' =>  
        404 ) );  
    }  
  
    return $posts[0]->post_title;  
}
```

As you can see, you create a custom function called `pdev_return_post_title_by_author_id()`, which accepts a `$data` parameter. You'll use the `get_posts()` function to query the posts within WordPress. You are passing in the author ID value via the `$args` parameter to return posts for a given author only. For security reasons, you wrap the author ID value in `absint()` to confirm that only an integer is passed through the function. If the `$posts` variable is empty, meaning `get_posts()` returned no results, you'll set a custom `WP_Error` notice to alert the system

that no posts matched with the author ID provided. Finally, you'll return the post title if a match is found.

Now that your function is in place to return the results, let's create a custom REST API endpoint to return the results. To do so, you'll utilize the `register_rest_route()` function, which accepts the following parameters:

- `$namespace`: A URL segment unique to your plugin.
- `$route`: The base URL for the route being added.
- `$args`: Optional array of options for your endpoint.
- `$override`: If the route already exists, should we override it?

If your namespace is `pdev-plugin/v1` and your route is `/author`, then the URL for it will be `/wp-json/pdev-plugin/v1/author/`.

NOTE *Namespaces should follow the pattern of `plugin/v1`, where `plugin` is generally your plugin or theme slug, and `v1` represents the first version of the API. If you ever need to break compatibility with new endpoints, you can then bump this to `v2`.*

Now let's register the custom endpoint for the example. You'll see the namespace, route, and an array of `$args` to define the method and callback function to use, as shown in the following code:

```
add_action( 'rest_api_init', 'pdev_custom_endpoint' );

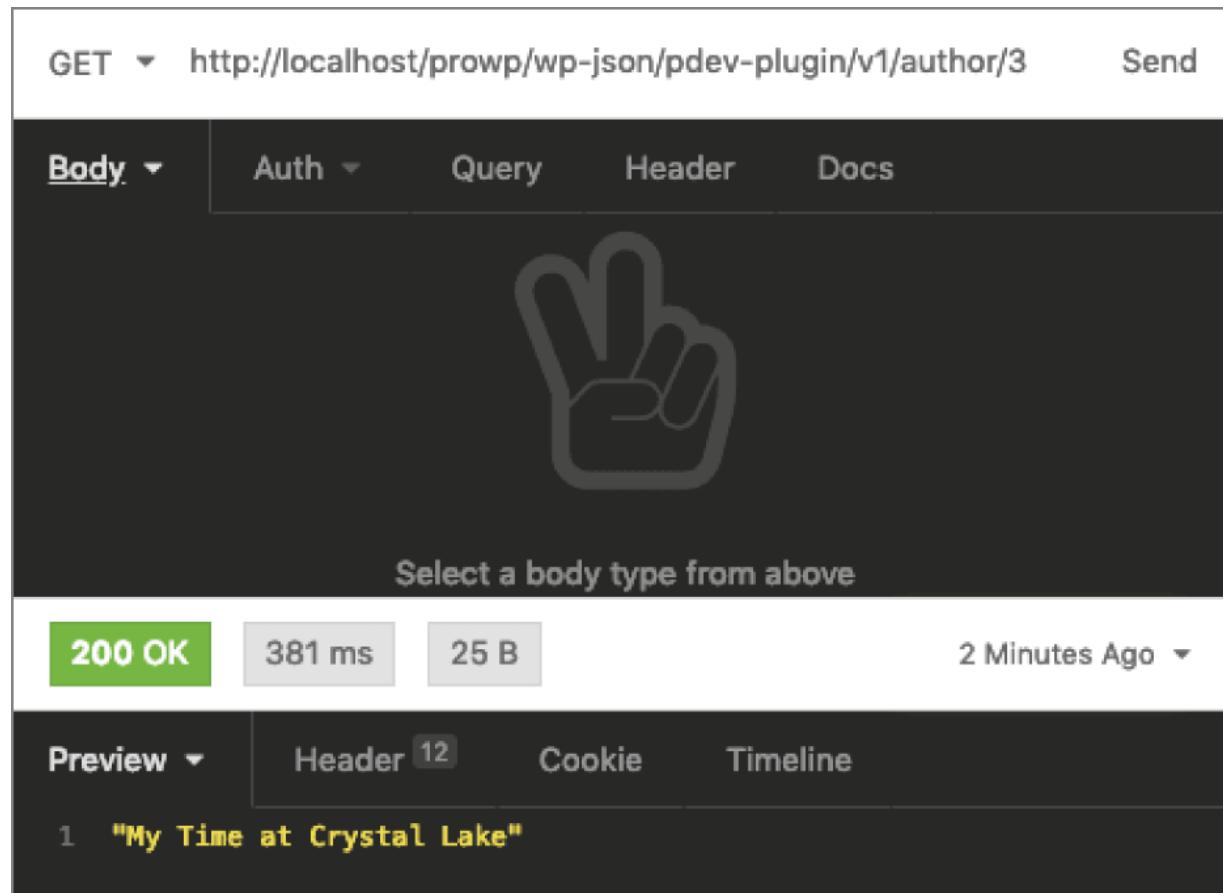
// Register your REST API custom endpoint
function pdev_custom_endpoint() {

    register_rest_route( 'pdev-plugin/v1', '/author/(?P<id>\d+)',
        array(
            'methods'  => 'GET',
            'callback' =>
            'pdev_return_post_title_by_author_id',
        )
    );
}
```

Now you have a custom REST API endpoint located here:

`http://localhost/prowp/wp-json/pdev-plugin/v1/author/<ID>`

where `<ID>` is the author ID whose post you want to retrieve. As a test, let's send a `GET` request to the new endpoint. In this example, you want to find the latest post from Jason Voorhees, who is user ID 3 in WordPress. In this example, the post titled "My Time at Crystal Lake" is returned, since that is the latest post published by Jason. You can see the results when you access the new custom endpoint, as shown in [Figure 12-6](#).



The screenshot shows a REST API client interface. At the top, a GET request is made to `http://localhost/prowp/wp-json/pdev-plugin/v1/author/3`. The response status is 200 OK, with a response time of 381 ms and a size of 25 B, timestamped 2 Minutes Ago. The response body is displayed as a single post: "1. 'My Time at Crystal Lake'". The interface includes tabs for Body, Auth, Query, Header, and Docs, and a preview section showing the post details.

[FIGURE 12-6](#): "My Time at Crystal Lake" post

Let's review the full REST API custom endpoint plugin, shown here:

```
<?php
/*
Plugin Name: PDEV REST API Custom Endpoint
Plugin URI: https://example.com/
Description: Register a custom endpoint in the WP REST API
```

```

Author: WROX
Author URI: http://wrox.com
 */

/**
 * Grab latest post title by the author ID
 *
 * @param array $data Options for the function.
 * @return string|null Post title for the latest, * or null
 * if none.
 */
function pdev_return_post_title_by_author_id( $data ) {

    $posts = get_posts( array(
        'author' => absint( $data['id'] ),
    ) );

    if ( empty( $posts ) ) {
        return new WP_Error( 'no_author', 'Invalid author',
array( 'status' =>
        404 ) );
    }

    return $posts[0]->post_title;
}

add_action( 'rest_api_init', 'pdev_custom_endpoint' );

// Register your REST API custom endpoint
function pdev_custom_endpoint() {

    register_rest_route( 'pdev-plugin/v1', '/author/(?P<id>\d+)',
        array(
            'methods' => 'GET',
            'callback' =>
    'pdev_return_post_title_by_author_id',
        )
    );
}

```

THE HTTP API

On the modern web, Internet-based services communicate with each other: web-based readers gather data from blog feeds and Twitter accounts, and personal websites display Facebook posts or YouTube videos.

Your site should be no exception to this interoperability. In this section, you'll learn how to make WordPress exchange information with the remote services API and open it to a whole new level of perspective.

WordPress features a number of functions that help make HTTP requests easily, called the **HTTP API**.

This section explains what exactly an HTTP request is, what it can be used for, and why you will once again thank WordPress for lending a convenient hand and doing the cumbersome parts for you.

What Is an HTTP Request?

Hyper Text Transfer Protocol (HTTP) is the networking protocol that is no less than the foundation of data communication for the World Wide Web.

HTTP Request Concepts

Even if you cannot name or explain the following concepts yet, you have already experienced them in your everyday life online, using a web browser. HTTP is a request/response protocol in the client/server computing model.

- **Client/server:** An application (the client) talks to another application (the server) that itself can respond to many clients at the same time. In the HTTP model, a client is, for example, a web browser, such as Firefox running on your computer, and the server is a web server powered, for instance, by Apache, PHP, and MySQL and running WordPress. A client can also be a web indexing spider robot or a PHP script that fetches and parses a web page to retrieve information. You do this later in this chapter.
- **Request/response protocol:** The client submits an HTTP request (basically “Hello, I'm Firefox, please send me file example.html”), and the server sends back a response (“Hello, I'm the Apache running PHP; here is the file, it is 4kb in size,” followed by the file itself). Both

requests contain potentially interesting information you learn to decipher and use.

Dissecting an HTTP Transaction

An HTTP transaction is a simple and clear-text communication between the client and the server.

The Client Sends a Request

The client request typically consists of a few lines sent in clear text to the server. Using Firefox as a web browser and trying to load <http://example.com/file.html> from a Google result page would translate into the following query:

```
GET /file.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0)
Firefox/3.6
Referer: https://www.google.com/search?q=example.com
Cookie: lastvisit=235456684
```

The first line starts with `GET`. A `GET` session is how you tell the server you want to retrieve a document, here `file.html` from host example.com. Other main requests methods you can use are `HEAD` (to just receive the server response headers) and `POST` (to submit data to a form).

Notice also how information such as the referrer URL or the user agent string is also sent by the client. In [Chapter 4](#), “Security and Performance,” you read that this data should not be trusted. Indeed, in an example later, you learn how to forge these values to anything.

The Server Sends a Response

The server response consists of three parts: the headers, with information about the response; a blank line; and then the response body.

The headers are a few lines of information and can be something like this:

```
HTTP/1.1 200 OK
Date: Mon, 31 October 2020 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Sun, 30 Oct 2020 23:11:55 GMT
Set-Cookie: lastvisit=235456951
```

Content-Length: 438

Content-Type: text/html; charset=UTF-8

The first interesting information is the status code, here 200. Each server response should have a status code giving details on how the transaction was handled by the server: 200 means OK, and 404 means not found. [Table 12-1](#) lists the main HTTP status codes you can use.

TABLE 12-1: Main HTTP Status Codes

Source: http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

STATUS CODE	SIGNIFICATION
200	OK
301	Moved Permanently
302	Moved Temporarily
403	Forbidden
404	Not Found
500	Internal Server Error
503	Service Unavailable

Of course, you don't have to memorize all these status codes, but with some experience you can quickly remember the main classes, as detailed in [Table 12-2](#).

TABLE 12-2: HTTP Status Code Classes

STATUS CODE	SIGNIFICATION
2xx	Request was successful.
3xx	Request was redirected to another resource (like in the case of a URL shortener).
4xx	Request failed because of a client error (for instance, a wrong username/password combination).
5xx	Request failed because of a server error (like a bad configuration or a broken script).

The server response also generally discloses information about the software running on the server, the content-type of the document it serves, and its length.

Some Caveats on Checking HTTP Responses

When you want to programmatically check the existence and validity of a link with an HTTP request, you can break your analysis down into two steps. If the request is successful and the response code is 404, you know the link does not exist. Otherwise, you may have to check things more carefully, depending on the context.

- If the request is an `is_wp_error()`, it can be because the URL to check is malformed but also because there is a temporary glitch preventing your web server from accessing the URL (connection problem, DNS timeout, and so on).
- If the response code is in the 5xx family (a server error; remember [Table 12-2](#)), this is probably a temporary server error, so you need to check again later.
- Some web servers are configured to handle “Not Found” errors differently than expected. For instance, <http://example.com/icons> will return a 404 when you would have expected the server to redirect to <http://example.com/icons/>, which actually exists.
- Some proxies or DNS servers, especially in corporate environments, are configured to handle all requests successfully, even though they should have returned an error. The following result is the actual return of `wp_remote_head('https://example.xom')` (notice the typo in the top-level domain) behind such a proxy, treating a nonexistent domain as a regular 404 error.

```
array(4) {
  ["headers"]=>
  array(6) {
    ["cache-control"]=>
    string(8) "no-cache"
    ["pragma"]=>
    string(8) "no-cache"
    ["content-type"]=>
    string(24) "text/html; charset=utf-8"
```

```
  ["proxy-connection"]=>
  string(10) "Keep-Alive"
  ["connection"]=>
  string(10) "Keep-Alive"
  ["content-length"]=>
  string(3) "762"
}
["body"]=>
string(0) ""
["response"]=>
array(2) {
  ["code"]=>
  int(404)
  ["message"]=>
  string(9) "Not Found"
}
["cookies"]=>
array(0) {
}
}
```

Possibilities for Crafting HTTP Requests

The first obvious use of HTTP requests is to retrieve a remote document or particular information within a remote document: a Twitter user's last message, the current value of share stock, or JSON-encoded data from a remote API service.

You can also send information to a remote document, such as a form or an HTTP API, and modify data from a client script.

These requests would be done using either `GET` or `POST` methods, sometimes with credentials (a login and password or another authentication mechanism) or other parameters. You can make such requests later in this chapter.

Another interesting application, using `HEAD` requests, is to check the state of a remote document without bothering to download its content. For instance, a broken link checker plugin could make sure your bookmarks in WordPress don't return a 404 header.

How to Make HTTP Requests in PHP

In basic PHP, without WordPress that is, there are several common ways to send HTTP requests. It is interesting to know the basics because you sometimes need to code a portion of code in a non-WordPress environment.

The following examples all do the same thing: send a GET request to <https://wordpress.org> and display the content received (that is, their index page).

Using the HTTP Extension

You can use the HTTP extension to send a GET request to <https://wordpress.org> and display the content received.

```
<?php

$r = new HttpRequest( 'https://wordpress.org/',
HttpRequest::METH_GET );
$r->send();
echo $r->getResponseBody();
?>
```

Using fopen() Streams

You can use fopen() streams to send a GET request to <https://wordpress.org/> and display the content received.

```
<?php

if( $stream = fopen( 'https://wordpress.org/', 'r' ) ) {
    echo stream_get_contents( $stream );
    fclose( $stream );
}
?>
```

Using a Standard fopen()

You can use a standard fopen() to send a GET request to <https://wordpress.org/> and display the content received.

```
<?php

$handle = fopen( 'https://wordpress.org/', 'r' );
$contents = '';
while( !feof( $handle ) ) {
    $contents .= fread( $handle, 8192 );
```

```
}

fclose( $handle );
echo $contents;
?>
```

Using fsockopen()

You can use `fsockopen()` to send a GET request to <https://wordpress.org/> and display the content received.

```
<?php

$fp = fsockopen( 'www.example.com', 80, $errno, $errstr, 30
);
if ( !$fp ) {
    echo "$errstr ($errno)<br/>\n";
} else {
    $out = "GET / HTTP/1.1\r\n";
    $out .= "Host: www.example.com\r\n";
    $out .= "Connection: Close\r\n\r\n";
    fwrite( $fp, $out );
    while ( !feof( $fp ) ) {
        echo fgets( $fp, 128 );
    }
    fclose( $fp );
}
?>
```

Using the CURL Extension

You can use the CURL extension to send a GET request to <https://wordpress.org/> and display the content received.

```
<?php

$ch = curl_init();

curl_setopt( $ch, CURLOPT_URL, 'https://wordpress.org/' );
curl_setopt( $ch, CURLOPT_HEADER, 0 );
curl_exec( $ch );
curl_close( $ch );
?>
```

Too Many Ways?

Each way has drawbacks and advantages over others. Some are simple and quicker to write, and some allow more parameters for finer control, support different request methods, or are faster to execute. Notice, for instance, how burdensome it is to use `fsockopen()`, which needs the complete request headers, compared to using streams or the HTTP extension.

The problem is this: depending on the server setup and configuration, PHP version, or security settings, some methods won't be allowed or even available. When working for a specific client, you could adapt to its specific server architecture and use a method you know will work, but this is simply impossible when authoring a plugin you intend to release for broad use.

What you should do, simply put, boils down to this alternative: either test each method prior to using one or rely on WordPress' HTTP API.

WORDPRESS' HTTP FUNCTIONS

WordPress implements a smart and powerful class, named `WP_Http` and found in `wp-includes/class-http.php`, that can test each previously described method and automatically select the best one available on the current machine.

The HTTP API supports all the methods you need to use (`GET`, `POST`, and `HEAD`) and enables fine-tuning several parameters such as proxy tunneling.

NOTE *Don't use PHP native methods to perform HTTP requests. Remember, they may be not installed or have restrictive configurations on many web hosts. Always use the WordPress HTTP API and its functions, described next.*

The `wp_remote_` Functions

You can execute an HTTP request within WordPress mostly using functions `wp_remote_get()`, `wp_remote_post()`, and `wp_remote_head()`, obviously for `GET`, `POST`, and `HEAD` requests.

These functions all operate the same way.

- The HTTP request is performed using the eponymous method.

- They accept two parameters, one required and one optional.
- They return an array or an object.

The syntax of these three functions follows:

```
<?php

$get_result = wp_remote_get( $url, $args );
$post_result = wp_remote_post( $url, $args );
$head_result = wp_remote_head( $url, $args );

?>
```

These three functions can be considered as simple shortcuts to the more generic `wp_remote_request()`. Indeed, the three preceding lines are equivalent to the three following ones:

```
<?php

$get_result = wp_remote_request( $url, array( 'method' =>
'GET' ) );
$post_result = wp_remote_request( $url, array( 'method' =>
'POST' ) );
$head_result = wp_remote_request( $url, array( 'method' =>
'HEAD' ) );

?>
```

The function `wp_remote_request()` works the same way as the other `wp_remote_*` functions, so everything that follows applies to any `wp_remote_` function.

You'll now learn what parameters they need and what data they return, and then you'll play with them.

`wp_remote_*` Input Parameters

The first parameter these functions need, `$url`, is a string representing a valid site URL to which the HTTP request will be sent. Supported protocols are HTTP and HTTPS; some transports might work with other protocols such as FTP, but don't assume this.

The second parameter, `$args`, is an optional array of parameters to override the defaults. The default parameters are the following array:

```

<?php

$defaults = array (
    'method' => 'GET',
    'timeout' => 5,
    'redirection' => 5,
    'httpversion' => '1.0',
    'user-agent' => 'WordPress/5.3; https://example.com/',
    'reject_unsafe_urls' => $url,
    'blocking' => true,
    'headers' => array (),
    'cookies' => array (),
    'body' => NULL,
    'compress' => false,
    'decompress' => true,
    'sslverify' => true,
    'sslcertificates' => ABSPATH . WPINC .
    '/certificates/ca-bundle.crt',
    'stream' => false,
    'filename' => NULL,
    'limit_response_size' => NULL
)
?>

```

This array contains the default values when omitted. For instance, if you want to disguise your HTTP request as one made by a generic browser instead of identifying your blog in the user-agent string, you would write the following:

```

<?php

$args = array(
    'user-agent' => 'Mozilla/5.0 (Macintosh; Intel Mac OS X
10.14; rv:72.0)
    Gecko/20100101 Firefox/72.0',
);

$result = wp_remote_get( $url, $args );

?>

```

In [Chapter 4](#), “Security and Performance,” you learned that despite its trustful name, the PHP-generated array `$_SERVER` should not be trusted. As you can see, it takes a single PHP line to forge and fake the content of, for example, `$_SERVER['HTTP_USER_AGENT']`.

[Table 12-3](#) contains a comprehensive description of the most important default values. You can consider the others either partially implemented, not always functional depending on the transport used, or simply of minor interest.

TABLE 12-3: Default Settings of wp_remote_ Functions Optional Parameters

PARAMETER	SIGNIFICATION
method	Either GET, POST, or HEAD. Some transports (the HTTP or the CURL extension, for instance) may accept other rarely used methods such as PUT or TRACE, but should not be assumed.
timeout	A number of seconds: how long the connection should stay open before failing when no response.
user-agent	The user-agent used to identify “who” is performing the request. Defaults to WordPress/ followed by the version of WordPress running and the URL of the blog issuing the request.
headers	An array of additional headers.
cookies	An array of cookie values passed to the server.
body	The body of the request, either a string or an array, which is data submitted to the URL.

wp_remote_* Return Values

All wp_remote_* functions return an array if the request has completed or an error object if it was unsuccessful.

Remote API Request Example

Now that you understand HTTP requests and the functions available for those requests, let's put it to use accessing a public API. In this example, you will use a public API from <https://sampleapis.com>, which is a great resource when learning to work with APIs. The website features several different APIs available to experiment with. Let's create a plugin that pulls horror movies from the Sample API service.

First you will register a custom menu for your plugin to display results from the API in the WordPress dashboard.

```
// Register a custom page for your plugin
add_action( 'admin_menu', 'pdev_create_menu' );

function pdev_create_menu() {
    // Create custom top-level menu
    add_menu_page( 'PDEV Movies Page', 'PDEV Movies',
        'manage_options', 'pdev-movies',
        'pdev_movie_api_results',
        'dashicons-smiley', 99 );
}

}
```

The preceding code registers your new menu and page, which then calls the `pdev_movie_api_results()` function. Next you'll register this function and use `wp_remote_get()` to retrieve movie data from the SampleAPI's Movie API. The remote request URL in this example is

<https://sampleapis.com/movies/api/horror>

```
// Request and display Movie API data
function pdev_movie_api_results() {

    // Set your API URL
    $request = wp_remote_get(
        'https://sampleapis.com/movies/api/horror' );
    // ... rest of the function code
}
```

It's always good practice to check whether an error is returned before working with the data. You'll do this using the `is_wp_error()` function and checking the returned `$request` variable, as shown here:

```
// If an error is returned, return false to end the request
if( is_wp_error( $request ) ) {
    return false;
}
```

Now that you've confirmed there are no errors, it's time to parse the data returned.

```
// Retrieve only the body from the raw response
$body = wp_remote_retrieve_body( $request );
```

```
// Decode the JSON string
$data = json_decode( $body );
```

First, you'll use the `wp_remote_retrieve_body()` function to retrieve only the body of the raw response. The only required parameter for this function is the HTTP response returned from your `wp_remote_get()` call. Next, you'll use the `json_decode()` PHP function, which takes a JSON-encoded string and converts it into a PHP variable.

You now have the Movie API data in a PHP array that you can loop through to display. The following code verifies the `$data` variable is not empty and then proceeds to loop through the array values:

```
// Verify the $data variable is not empty
if( ! empty( $data ) ) {

    echo '<ul>';

    // Loop through the returned dataset
    foreach( $data as $movies ) {

        echo '<li>';
        echo '<a href="https://www.imdb.com/title/' . 
        esc_attr( $movies->imdb_id ) . '">';
        echo '<br/>';
        echo esc_html( $movies->title );
        echo '</a>';
        echo '</li>';

    }

    echo '</ul>';
}
```

You now have a working plugin that pulls horror movie data from the [SampleAPIs.com](#) service. The movie image and title are displayed in a list, with a link out to IMDB to view more details about each movie. You also incorporated proper escaping functions for enhanced security, since you should never trust data coming from a third-party system. This is a simple example of how to access an external API and work with the data it returns. Let's review the full plugin source code shown here:

```
<?php
/*
Plugin Name: Horror Movie API Example Plugin
Plugin URI: https://example.com/
Description: Example using the HTTP API to parse JSON from a
remote
horror movie API
Author: WROX
Author URI: http://wrox.com
*/
// Register a custom page for your plugin
add_action( 'admin_menu', 'pdev_create_menu' );

function pdev_create_menu() {

    // Create custom top-level menu
    add_menu_page( 'PDEV Movies Page', 'PDEV Movies',
        'manage_options', 'pdev-movies',
        'pdev_movie_api_results',
        'dashicons-smiley', 99 );

}

// Request and display Movie API data
function pdev_movie_api_results() {

    // Set your API URL
    $request = wp_remote_get(
        'https://sampleapis.com/movies/api/horror' );

    // If an error is returned, return false to end the
    request
    if( is_wp_error( $request ) ) {
        return false;
    }

    // Retrieve only the body from the raw response
    $body = wp_remote_retrieve_body( $request );

    // Decode the JSON string
    $data = json_decode( $body );

    // Verify the $data variable is not empty
    if( ! empty( $data ) ) {

        echo '<ul>';

    }

}

// Add the custom page to the admin menu
function pdev_add_menu() {
    add_menu_page( 'PDEV Movies Page', 'PDEV Movies',
        'manage_options', 'pdev-movies',
        'pdev_movie_api_results',
        'dashicons-smiley', 99 );
}

// Register the custom menu
add_action( 'admin_menu', 'pdev_add_menu' );

```

```

// Loop through the returned dataset
foreach( $data as $movies ) {

    echo '<li>';
    echo '<a href="https://www.imdb.com/title/' . esc_attr( $movies->imdb_id ) . '">';
    echo '<br/>';
    echo esc_html( $movies->title );
    echo '</a>';
    echo '</li>';

}

echo '</ul>';
}
}

```

NOTE *Remember when retrieving data from a remote API it's important to always consider caching the results. Given the previous code example, you wouldn't want to hit the API on every page load. Cache the results and refresh that cache as often as it makes sense to do so. You can review different caching techniques in [Chapter 4](#), “Security and Performance.”*

Unsuccessful Requests

In case of a malformed HTTP request or if the request cannot be performed for any other reason (site not responding, temporary connection problem, etc.), the result will be an object instance of WordPress' class `WP_Error`, containing an error code and an error message, as illustrated in the following code snippet:

```

<?php

var_dump( wp_remote_get( 'malformed-url' ) );

?>

```

The result of this ill-fated GET request follows:

```
object(WP_Error)#1212 (2) {
    ["errors"]=>
    array(1) {
        ["http_request_failed"]=>
        array(1) {
            [0]=>
            string(29) "A valid URL was not provided."
        }
    }
    ["error_data"]=>
    array(0) {
    }
}
```

Error objects returned by HTTP requests will contain the error code `http_request_failed` and a meaningful detailed diagnosis. Consider the following attempts:

```
<?php
$bad_urls = array(
    'malformed',
    'irc://example.com/',
    'http://inexistent',
);
foreach( $bad_urls as $bad_url ) {

    $response = wp_remote_head( $bad_url, array( 'timeout' => 1 ) );

    if( is_wp_error( $response ) ) {

        $error = $response->get_error_message();
        echo "<p>Trying $bad_url returned: <br/> $error
</p>";
    }
}
?>
```

Notice a couple of things in this snippet:

- To speed up things because it's obvious these requests will fail and you don't want to wait for 5 seconds each, an additional timeout parameter is set to 1 second.
- Because HTTP requests return a `WP_Error` object on failure, you can test the response using function `is_wp_error()`. You'll learn more about dealing with errors and the `WP_Error` class in [Chapter 15](#), “Debugging.”

Finally, look at the actual result of this code snippet:

```
Trying malformed returned:  
A valid URL was not provided.
```

```
Trying irc://example.com/ returned:  
A valid URL was not provided.
```

```
Trying http://inexistent returned:  
CURL error 6: Couldn't resolve host 'inexistent'
```

As you can see, the HTTP request functions can diagnose most scenarios, so you know you can rely on them if you need to troubleshoot unexpected behavior within your code.

Successful Requests

When the HTTP request has completed, `wp_remote_` functions return a multidimensional array of four elements, containing the raw server response in four parts: ‘headers’, ‘body’, ‘response’, and ‘cookies’.

Consider the following request:

```
<?php  
var_dump( wp_remote_get( 'http://example.com/asdfgh' ) );  
?>
```

The output of this request will be akin to the following:

```
array(4) {  
  ["headers"] => array(5) {  
    ["date"] => string(29) "Thu, 16 Jan 2020 20:04:01 GMT"
```

```

[“server”] => string(85) “Apache/2.4.41 mod_ssl/2.2.8
PHP/7.3”
[“content-length”] => string(3) “648”
[“connection”] => string(5) “close”
[“content-type”] => string(25) “text/html; charset=utf-
8”
}

[“body”] => string(1256) “<html><head>
<title>404 Not Found</title>
</head><body>
(... snip ...)
</body></html>
”

[“response”] => array(2) {
    [“code”] => int(404)
    [“message”] => string(9) “Not Found”
}

[“cookies”] => array(0) {}
}

```

The first thing you should note here is that despite sending an HTTP request to a nonexistent page, the request is still considered successful. Whenever the web server replies to the client request, no matter its reply, the HTTP transaction is complete.

The four primary elements of the response array consist of the following:

- **headers**: The raw list of the server response as detailed in the first section of this chapter, minus the HTTP response code.
- **body**: The body of the server response, which is typically the page HTML content itself but can be JSON- or XML-encoded data when polling a remote API, for instance.
- **response**: The server response code and its signification, as detailed in [Table 12-1](#) and [Table 12-2](#). This particular information is especially valuable. Despite the HTTP transaction being successful, its result may be totally different from what you expect. You should always check that you obtain 200 as a response code.
- **cookies**: If the server wants the client to store cookie information, that information will be included here. In case you need this information

for any subsequent HTTP request, include the information as an additional optional parameter in the next `wp_remote_` function call.

wp_remote_ Companion Functions

The array returned by `wp_remote_` functions contains exhaustive information and as such may contain too much data if you need just a part of it.

Along with functions performing HTTP requests, you can use “companion” functions that enable quick access to a part of the returned array.

- `wp_remote_retrieve_response_code()`: Returns just the response code (for example, 200) of an HTTP response
- `wp_remote_retrieve_response_message()`: Returns just the response message (for example, “OK”)
- `wp_remote_retrieve_body()`: Returns the body of the response
- `wp_remote_retrieve_headers()`: Returns all the headers of a server response
- `wp_remote_retrieve_header()`: Returns just one particular header from a server response

For example, to check if a link exists and does not return a 404 Not Found error, you can use the following code:

```
<?php

$url = 'https://www.example.com/bleh';

// Send GET request
$response = wp_remote_get( $url );

// Check for server response
if( is_wp_error( $response ) ) {

    $code = $response->get_error_message();
    wp_die( 'Requests could not execute. Error was: ' .
$code );

}
```

```
// Check that the server sent a "404 Not Found" HTTP status
code
if( wp_remote_retrieve_response_code( $response ) == 404 ) {

    wp_die( 'Web page not found' );

}

// So far, so good
echo 'Web page found!';

?>
```

You use these simple companion functions more in the next examples and plugins.

Advanced Configuration and Tips

Thanks to these `wp_remote_` functions, you are now able to perform most tasks involving HTTP requests in a standard WordPress environment. But not all environments are customary, and not all tasks are basic. Fortunately, the HTTP API is extensible and versatile.

For instance, it is frequent that networks in corporate environments are isolated behind a firewall or a proxy. You will now read how to bypass this and maybe treat HTTP responses differently.

In the following sections, you will also learn how to fine-tune the behavior of the HTTP API, utilizing its hooks and filters, for example to log requests for troubleshooting.

Proxy Support

In computer networks, a proxy server is a server that acts as an intermediary between the client and the requested server.

A great aspect of the HTTP API, and another reason why it is superior to PHP native functions as detailed earlier, is that it supports connections through a proxy without additional complex configuration.

To enable proxy support, you simply need to have the user define the following constants:

```
<?php

define( 'WP_PROXY_HOST', 'firewall.corp.example.com' );
define( 'WP_PROXY_PORT', '3128' );
define( 'WP_PROXY_USERNAME', 'username' );
define( 'WP_PROXY_PASSWORD', 'password' );

?>
```

This is especially important for users in a corporate environment where proxies are common and can block all WordPress' outgoing requests if not, or incorrectly, configured.

On a corporate network, where a firewall architecture can characteristically handle different connections toward the Internet and those staying on the intranet, another constant can be used to specify domains that should not go through the proxy, in a comma-separated list.

```
<?php

// these hosts will not go through the proxy
define( 'WP_PROXY_BYPASS_HOSTS', 'sales.example.com,
hr.example.com' );

?>
```

The blog domain and `localhost` are automatically added to this list, so you don't have to include them. Wildcards using `*` are supported, as in `*.wordpress.org`.

Also, when working with clients on a firewalled corporate intranet, a concern of your client's IT department may be to limit outgoing connections to a restricted white list of websites. If so, use constants `WP_HTTP_BLOCK_EXTERNAL` and `WP_ACCESSIBLE_HOSTS` like so:

```
<?php

// block all requests through the HTTP API
define( 'WP_HTTP_BLOCK_EXTERNAL', true );

// except for these hosts
define( 'WP_ACCESSIBLE_HOSTS',
    'api.wordpress.org, sales.example.com, partner.web' );
?>
```

Including api.wordpress.org in the list of accessible hosts can ensure that the built-in upgrading for core, plugins, and themes still work.

Filtering Requests and Responses

As any other piece of WordPress code poetry, the HTTP API makes considerable use of hooks, and by reading the source file of the `WP_Http` class, you can find several filters and actions triggered.

Example: Modify a Default Parameter

For instance, if you want all your plugins to show off your WordPress skills in server logs whenever they perform queries, add the following filter and function:

```
<?php

// Hook into the filter that sets user agent for HTTP
// requests
add_filter( 'http_headers_useragent',
'pdev_plugin_user_agent' );

// Set your own user agent
function pdev_plugin_user_agent() {
    global $wp_version;

    return "WordPress version $wp_version ; ".
        "Need a WordPress specialist? Contact us! ".
        "PDEV Studios www.example.com";

}

?>
```

This filter can set the new default value for the user agent string, which means that on a per-request basis you can still override it, as in the previous example where you disguised it as a generic Internet browser.

Example: Log HTTP Requests and Responses

Hooks that can come in handy when debugging requests and server responses are the `http_request_args` and `http_response` filters, used to allow modification of the request's parameters right before the request is executed or just before the server responses are returned.

In the `WP_Http` class source (located in `wp-includes/class-http.php`), you can see that each request applied these two filters:

```
<?php

// before the request is sent, you will find:
$parsed_args = apply_filters( 'http_request_args',
$parsed_args, $url );

// once the response is processed, you will read:
return apply_filters( 'http_response', $response,
$parsed_args, $url );

?>
```

You are now going to code a plugin that logs each HTTP request and its parameters and each server response into a flat text file. You can use `pdev_http` as a prefix throughout this plugin.

```
<?php
/*
Plugin Name: PDEV Log HTTP requests
Plugin URI: https://example.com/
Description: Log all HTTP requests into a flat text file for
further analysis
Author: WROX
Author URI: http://wrox.com
*/

// Hook into filters
add_filter( 'http_request_args', 'pdev_http_log_request',
10, 2 );
add_filter( 'http_response', 'pdev_http_log_response', 10, 3
);

// Log requests.
// Parameters passed: request parameters and URL
function pdev_http_log_request( $r, $url ) {

    // Get request parameters formatted for display
    $params = print_r( $r, true );

    // Get date with format 2010-11-25 @ 13:37:00
    $date = date( 'Y-m-d @ H:i:s' );

    // Message to log:
```

```

$log = <<<LOG
$date: request sent to $url
Parameters: $params
-----
LOG;

// Log message into flat file
error_log( $log, 3, dirname( __FILE__ ).'/http.log' );

// Don't forget to return the requests arguments!
return $r;
}

// Log responses
// Parameters passed: server response, requests parameters
// and URL
function pdev_http_log_response( $response, $r, $url ) {

    // Get server response formatted for display
    $resp = print_r( $response, true );

    // Get date with format 2020-10-31 @ 13:37:00
    $date = date( 'Y-m-d @ H:i:s' );

    // Message to log:
    $log = <<<LOG
$date: response received from $url
Response: $resp
-----
LOG;

    // Log message into flat file
    error_log( $log, 3, dirname( __FILE__ ).'/http.log' );

    // Don't forget to return the response!
    return $response;
}

?>

```

The two logging functions are similar. They receive from the filters several parameters that are then printed into a flat text file using PHP function `error_log()`; then they eventually return the unmodified filtered value.

Notice the syntax used here to delimit strings, called the *heredoc syntax*. The opening string delimiter is an identifier after <<<, and the closing delimiter is the identifier, not indented.

After you activate this plugin, it starts appending entries to the file `http.log` in the plugin's directory. This is an interesting plugin that demonstrates the inner working of WordPress' core, because it will, for instance, log all transactions with api.wordpress.org when checking the latest version of plugins, themes, and core, or when fetching the feeds displayed in your dashboard.

NOTE *Remember that logging events is for debugging only and is not suitable for production environments, as it could leak sensitive information or even fill up disk space with log data.*

BRINGING IT ALL TOGETHER

Now that you've learned about the primary HTTP API functions available and have a good understanding of HTTP requests, let's put it all together using the HTTP API to interact with the WordPress REST API. To do this, I recommend you create two test websites. One website will run the plugin to send requests to the REST API of the second website. Another option is to create a local Multisite setup with two sites in the network, as covered in [Chapter 13](#), "Multisite."

Create

Let's look at an example that creates a new post on an external WordPress website using the REST API. First, let's register a basic menu page in the Dashboard.

```
// Register a custom page for your plugin
add_action( 'admin_menu', 'pdev_create_menu' );

function pdev_create_menu() {

    // Create custom top-level menu
    add_menu_page( 'PDEV REST API FUN', 'PDEV REST API',
        'manage_options', 'pdev-rest-api',
```

```
'pdev_create_new_post',
'dashicons-smiley' );
}
```

Now you have an area to work with your code. Next, let's create a simple form that when submitted will trigger the request to create a new post.

```
<h1>Create a Post using the REST API</h1>
<p>Click the button below to create a new post on an
external WordPress
website using the REST API</p>
<form method="post">
    <input type="submit" name="submit" class="button-
primary"
        value="Create Post"/>
</form>
```

As you can see, there is nothing special here, just a simple HTML form. Now comes the fun part, defining the data you intend to send to the REST API for your new post, as shown here:

```
if ( isset( $_POST['submit'] ) ) {

    // Set the API URL to send the request
    $api_url = 'http://example.com/wp-json/wp/v2/posts';

    // Using Basic Auth, set your username and password
    $api_header_args = array(
        'Authorization' => 'Basic ' . base64_encode(
        'brad:pa55w0rd' )
    );

    // Create the new post data array
    $api_body_args = array(
        'title'    => 'REST API Test Post',
        'status'   => 'draft',
        'content'  => 'This is my test post. There are many
like it, but this
one is mine.',
        'excerpt'  => 'Read this amazing post'
    );
}
```

First you check whether the form submit button has been posted, using the `isset()` PHP function, and if the form has been submitted, we know it's time to create the new post. Next, you'll create variables to hold the new

post information. The first variable is `$api_url`, which contains the URL to the API of the WordPress website you plan to create your new post on. This needs to be a different website from the one running the plugin you are creating.

Next, you'll define the `$api_header_args` array, which sends the basic auth credentials to the REST API. Remember, for basic auth to work, the website you are creating the new post on needs to be running the Basic Auth plugin located at <https://github.com/WP-API/Basic-Auth>. Simply replace the `USERNAME` and `PASSWORD` text with a real username and password to authenticate.

Finally, you'll create the `$api_body_args` array, which contains all of the data for your new post. In this example, you set the title, content, and excerpt fields. You are also setting the post to a draft status so it is not published publicly yet.

Now that all the request data has been set, it's time to fire off the request to the remote REST API. You'll use the `wp_remote_post()` HTTP API function to do this. You'll notice you are passing in the three variables you set previously: `$api_url`, `$api_header_args`, and `$api_body_args`.

```
// Send the request to the remote REST API
$api_response = wp_remote_post( $api_url, array(
    'headers' => $api_header_args,
    'body' => $api_body_args
) );

// Decode the body response
$body = json_decode( $api_response['body'] );

// Verify the response message was 'created'
if( wp_remote_retrieve_response_message( $api_response ) ===
'Created' ) {
    echo '<div class="notice notice-success is-
dismissible">';
    echo '<p>The post ' . $body->title->rendered . ' has
been created
        successfully</p>';
    echo '</div>';
}
```

Once the request has been sent, you'll use `json_decode()` to decode the body response from the API. Using the

`wp_remote_retrieve_response_message()` function, you can quickly check whether the response message is `Created`, which means the post was successfully created, and show a success message. That's it! You have just successfully created a new post on a remote WordPress website using the REST API!

NOTE *When you're working with APIs, it's always important to include some level of error handling, as you reviewed earlier in the “Unsuccessful Requests” section of this chapter. Security should also be enhanced using a nonce, covered in [Chapter 4](#), “Security and Performance,” to verify the request is legitimate.*

Update

Now that you have created a new post using the REST API, let's send a request to update the post. To do this, you'll use much of the same code as before, with a few minor adjustments.

```
// Set the API URL to send the request
$api_url = 'http://example.com/wp-json/wp/v2/posts/<ID>';

// Using Basic Auth, set your username and password
$api_header_args = array(
    'Authorization' => 'Basic ' . base64_encode(
USERNAME:PASSWORD' )
);

// Create the post data array to update
$api_body_args = array(
    'title'    => 'UPDATED: REST API Test Post'
);

// Send the request to the remote REST API
$api_response = wp_remote_post( $api_url, array(
    'headers' => $api_header_args,
    'body'    => $api_body_args
) );

// Decode the body response
$body = json_decode( $api_response['body'] );

// Verify the response message was 'created'
```

```

if( wp_remote_retrieve_response_message( $api_response ) ===
'OK' ) {
    echo '<div class="notice notice-success is-
dismissible">';
    echo '<p>The post ' . $body->title->rendered . ' has
been updated
        successfully</p>';
    echo '</div>';
}

```

First the `$api_url` needs to be updated to include the unique post ID you want to update. Simply replace `<ID>` with the post ID in the previous code sample. You'll pass the same basic auth credentials in your `$api_header_args` array. The `$api_body_args` array will include any data you want to update. In this example, you are going to update the post title, so set that to something different from the original.

Next, you'll send the request and decode the body response just as before. The final change is to verify the response message is OK, meaning the post was updated. That's it! When executing the code, the post ID defined in the request will receive an updated title.

Delete

You have successfully created and updated a post. The final step is to delete a post using the REST API.

```

// Set the API URL to send the request
$api_url = 'http://example.com/wp-json/wp/v2/posts/<ID>/';

// Using Basic Auth, set your username and password
$api_header_args = array(
    'Authorization' => 'Basic ' . base64_encode(
    'USERNAME:PASSWORD' )
);

// Send the request to the remote REST API
$api_response = wp_remote_post( $api_url, array(
    'method' => 'DELETE',
    'headers' => $api_header_args
) );

// Decode the body response
$body = json_decode( $api_response['body'] );

```

```

// Verify the response message was 'created'
if( wp_remote_retrieve_response_message( $api_response ) ===
'OK' ) {

    echo '<div class="notice notice-success is-
dismissible">';
    echo '<p>The post ' . $body->title->rendered . ' has
been deleted
        successfully</p>';
    echo '</div>';

}

}

```

The `$api_url` will match the update request, which requires a post ID to be set. You'll notice you are setting the method to `DELETE` in the `wp_remote_post()` call. This is what tells the REST API to delete the post based on the ID in the `$api_url`. The code will not successfully trash the post when executed.

Now let's review the entire plugin to create, update, and delete a post using the WordPress REST API. Remember to update the `$api_url` variable as well as the `USERNAME` and `PASSWORD` fields for basic auth to work.

```

<?php
/*
Plugin Name: REST API - Create, Update, and Delete Post
Examples
Plugin URI: https://example.com/
Description: Create, update, and delete a new post using the
WordPress REST API
Author: WROX
Author URI: http://wrox.com
*/

```

```

// Register a custom page for your plugin
add_action( 'admin_menu', 'pdev_create_menu' );

function pdev_create_menu() {

    // Create custom top-level menu
    add_menu_page( 'PDEV REST API FUN', 'PDEV REST API',
        'manage_options', 'pdev-rest-api',
        'pdev_create_new_post',
        'dashicons-smiley' );

```

```

}

```

```
function pdev_create_new_post() {

    if ( isset( $_POST['create-post'] ) ) {

        // Set the API URL to send the request
        $api_url = 'http://example.com/wp-json/wp/v2/posts';

        // Using Basic Auth, set your username and password
        $api_header_args = array(
            'Authorization' => 'Basic ' . base64_encode(
                'USERNAME:PASSWORD'
            );

        // Create the new post data array
        $api_body_args = array(
            'title' => 'REST API Test Post',
            'status' => 'draft',
            'content' => 'This is my test post. There are
many like it, but this
                                one is mine.',
            'excerpt' => 'Read this amazing post'
        );

        // Send the request to the remote REST API
        $api_response = wp_remote_post( $api_url, array(
            'headers' => $api_header_args,
            'body' => $api_body_args
        ) );

        // Decode the body response
        $body = json_decode( $api_response['body'] );

        // Verify the response message was 'created'
        if( wp_remote_retrieve_response_message(
            $api_response ) === 'Created' ) {
            echo '<div class="notice notice-success is-
dismissible">';
            echo '<p>The post ' . $body->title->rendered . '
has been created
                                successfully</p>';
            echo '</div>';
        }

    }elseif ( isset( $_POST['update-post'] ) ) {

        // Set the API URL to send the request
        $api_url = 'http://example.com/wp-
```

```
json/wp/v2/posts/<ID>/';

// Using Basic Auth, set your username and password
$api_header_args = array(
    'Authorization' => 'Basic ' . base64_encode(
'USERNAME:PASSWORD' )
);

// Create the post data array to update
$api_body_args = array(
    'title' => 'UPDATED: REST API Test Post'
);

// Send the request to the remote REST API
$api_response = wp_remote_post( $api_url, array(
    'headers' => $api_header_args,
    'body' => $api_body_args
) );

// Decode the body response
$body = json_decode( $api_response['body'] );

// Verify the response message was 'created'
if( wp_remote_retrieve_response_message(
$api_response ) === 'OK' ) {
    echo '<div class="notice notice-success is-
dismissible">';
    echo '<p>The post ' . $body->title->rendered . ' '
has been updated
    successfully</p>';
    echo '</div>';
}

}elseif ( isset( $_POST['delete-post'] ) ) {

    // Set the API URL to send the request
    $api_url = 'http://example.com/wp-
json/wp/v2/posts/<ID>/';

    // Using Basic Auth, set your username and password
    $api_header_args = array(
        'Authorization' => 'Basic ' . base64_encode(
'USERNAME:PASSWORD' )
    );

    // Send the request to the remote REST API
    $api_response = wp_remote_post( $api_url, array(
        'method' => 'DELETE',
    )
);
```

```

        'headers' => $api_header_args
    ) );
}

// Decode the body response
$body = json_decode( $api_response['body'] );

// Verify the response message was 'created'
if( wp_remote_retrieve_response_message(
$api_response ) === 'OK' ) {

    echo '<div class="notice notice-success is-
dismissible">';
    echo '<p>The post ' . $body->title->rendered . '
has been deleted
        successfully</p>';
    echo '</div>';

}

?>
<h1>Create or Update a Post using the REST API</h1>
<p>Click the button below to create a new post on an
external WordPress
    website using the REST API</p>
<form method="post">
    <input type="submit" name="create-post"
class="button-primary"
        value="Create Post"/>
    <input type="submit" name="update-post"
class="button-primary"
        value="Update Post"/>
    <input type="submit" name="delete-post"
class="button-primary"
        value="Delete Post"/>
</form>
<?php
}

}

```

Resources

There are some great resources available for working with the REST and HTTP APIs. When looking for online help, it's important to look for newer articles and tutorials. Older examples, even from a year ago, could be using out-of-date methods.

The following are some resources to help you grow your knowledge on the APIs covered in this chapter:

- **REST API Handbook:** Official WordPress project handbook
 - <https://developer.wordpress.org/rest-api>
- **HTTP API Handbook:** Official WordPress project handbook
 - <https://developer.wordpress.org/plugins/http-api>
- **Kinsta Blog:** Extensive overview of the WordPress REST and HTTP API
 - <https://kinsta.com/blog/wordpress-rest-api>
 - <https://kinsta.com/blog/wordpress-http-api>

SUMMARY

It's easy to see how powerful the REST API in WordPress is. It truly transforms WordPress from a publishing platform to a true application framework that can be used for all sorts of interesting setups. This opens the door to endless possibilities when working with modern web applications and systems.

13

Multisite

WHAT'S IN THIS CHAPTER?

- Introduction to Multisite
- Unique Terminology
- Enabling & Configuring
- Common Functions
- Creating New Sites
- Switching between Sites
- Network and Site Options and Meta
- Users and Roles
- Database Schemas
- Query and Object Classes

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are at

www.wiley.com/go/prowordpressdev2e on the Downloads tab.

Multisite is an extremely powerful hidden feature that comes included with WordPress. Disabled by default, it is not the kind of feature that everyone will want to use and requires some additional configuration to make work smoothly.

Other projects and platforms often refer to this type of thing as *multitenancy*. Multisite allows any WordPress installation to be responsible for easily managing an unlimited number of sites inside it, traditionally called a *network* of sites. This network can have open registration of users, sites, or both or neither. There are a ton of specialty Multisite plugins to help customize everything, from signups to domain aliasing and redirection.

Each site in your Multisite network can have separate plugins and themes activated. They each are responsible for their own unique content, and user roles and capabilities are still assigned on a per-site basis.

As a plugin developer, you will need to understand what features are available when working with Multisite in WordPress so you know if and how to best support those features in the plugins you build.

Just because Multisite is built in to WordPress does not mean that it is identical to everything you are already familiar with. There are quite a few little things that make it just different enough to require planning for, and you don't want your plugins to not work correctly for someone who happens to be activating them in a Multisite environment.

TERMINOLOGY

The most important new terms in Multisite are *global*, *installation*, *network*, and *site*. A *site* is a single site inside a network of sites. A *network* is a group of sites. The *installation* is everything together. *Global* generally refers to users and user metadata but may also refer to Must-Use plugins, database drop-ins, or anything else that spans the entire environment.

When you first activate Multisite, you will have one network and one site in your installation. You will have unrestricted access to the entire installation, including the ability to create new sites. If you want to have multiple networks within your installation, you will need to install a plugin to enable this additional hidden functionality.

When developing plugins for Multisite, you need to determine whether you want them to work across the entire installation, across a single network, or only in a single site. For example, you may want to retrieve posts from one site in the network to be used inside others, or you may want to create a network-wide option for your plugin.

All sites in your installation have a status. The status is important and can determine whether the site is viewable by the public. The following are the available site statuses in Multisite:

- **Public:** The site is public if the privacy setting is set to enable search engines.

- Archived: The site has been archived and is not available to the public.
- Mature: The site is flagged as being for a mature audience.
- Spam: The site is considered spam and is not available to the public.
- Deleted: The site is flagged for deletion and is not available to the public.

The only two statuses that don't remove the site from public viewing are **Public** and **Mature**. **Mature** can be used if you want to allow mature sites in your network but need a way to warn users prior to them viewing the content. **Public** is based on the privacy settings and whether search engines are allowed to index the site.

Part of setting up Multisite means having to choose how domains for sites will be accessed: either as subdomains (site1.example.com) or as subdirectories ([example .com/site1](http://example.com/site1)). You can even set a fully qualified domain name for each site (example1.com) so that visitors to your sites will have no idea they are all powered by a single install of WordPress. Future versions of WordPress and Multisite plan to remove this strict requirement, as servers can now be configured to accept generic requests allowing for WordPress to identify sites from any configuration type.

As you can imagine, this is an extremely powerful feature in WordPress. There is no limit to the number of sites WordPress can run; the only restriction is the resources available on your hosting environment. WordPress.com is the largest Multisite installation of WordPress, powering millions of sites together in one giant installation.

ADVANTAGES OF MULTISITE

Running Multisite for your websites offers many advantages. The most obvious one is you have only a single install of WordPress to manage. This makes life much easier when keeping everything up to date. If you have 50 sites and a plugin update is released, updating it affects all sites in your installation. If each site were a separate install of WordPress, you would have to update the plugin 50 times.

Another advantage to Multisite is the ease with which you can aggregate content across your network. For example, if you have 50 sites in your network, you could easily aggregate all those posts to your main blog to showcase your network of sites. If the sites were separate installs of WordPress, it would take quite a bit more work to aggregate that content.

Managing sites in Multisite is also versatile. You can easily limit disk space usage on each site. You can dictate what file types are allowed for uploading along with file size limits. You can even lock down plugins and themes from being activated by the other site administrators.

ENABLING MULTISITE IN WORDPRESS

Installing WordPress Multisite is straightforward. One of the great features of Multisite is that it can be enabled prior to installing WordPress, or anytime thereafter. If you decide to convert your WordPress site into Multisite a year down the road, you can easily do so.

The first step to enabling Multisite is to modify your `wp-config.php` file. This file contains your database connection settings and other important configuration options. To enable Multisite, you need to add the following line before where it says `/* That's all, stop editing! Happy blogging. */`:

```
define( 'WP_ALLOW_MULTISITE', true );
```

Adding this line to your `wp-config.php` file enables the Tools \Rightarrow Network menu options, as shown in [Figure 13-1](#).

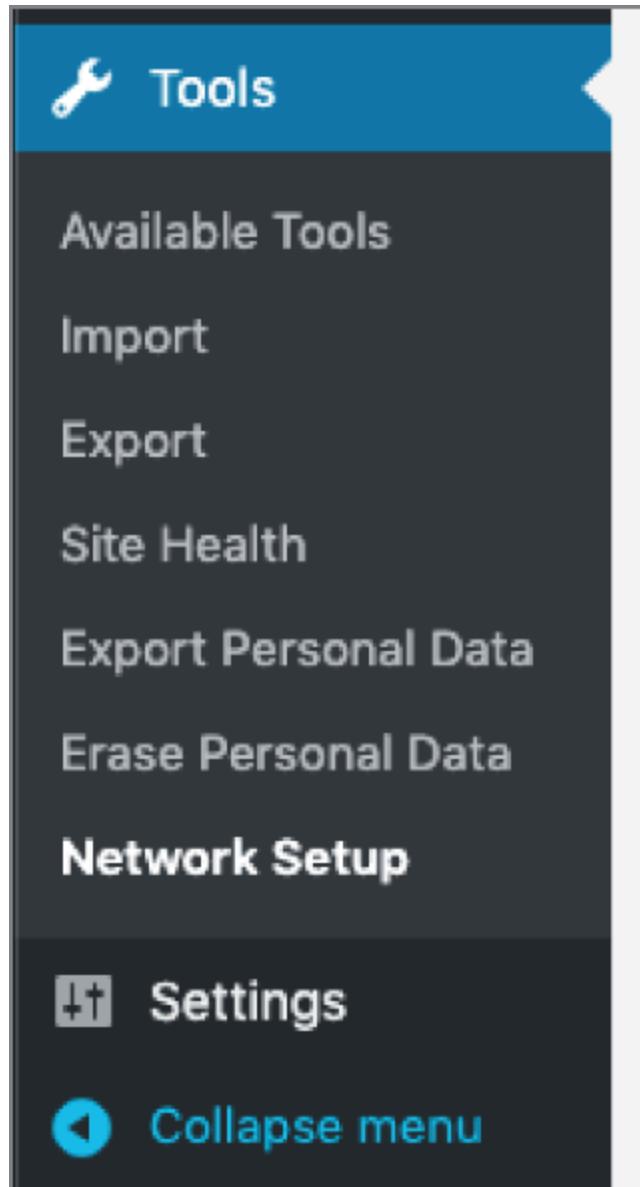


FIGURE 13-1: Tools ⇔ Network menu options

Visiting this new menu option takes you to the Create a Network admin page. If you have not done so already, you will be required to disable all plugins prior to enabling Multisite. This must be done to ensure that nothing unexpected is happening while the installation process is running.

Here you can find detailed instructions on the necessary steps to complete the Multisite installation. In this tutorial, you will configure Multisite to work with subdirectories, so if you plan to use subdomains, be sure to follow the installation instructions closely as the code may differ slightly.

If you are unsure whether to use subdomains or subdirectories, we strongly recommend starting with subdirectories, as the lookups and pathing for finding sites are much more flexible. If this feature ever gets removed, subdirectories will be the default installation type.

Next you need to add the following code to your `wp-config.php` file. Note that this is example code, and the `DOMAIN_CURRENT_SITE` constant would contain your website's domain in place of example.com.

```
$base = '/';
define( 'MULTISITE', true );
define( 'SUBDOMAIN_INSTALL', false );
define( 'DOMAIN_CURRENT_SITE', 'example.com' );
define( 'PATH_CURRENT_SITE', '/' );
define( 'SITE_ID_CURRENT_SITE', 1 );
define( 'BLOG_ID_CURRENT_SITE', 1 );
```

If you are using Apache for your web server, the final step is to modify your `.htaccess` file in the root directory of your WordPress installation. Replace the existing WordPress rules with the following code:

```
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]

# uploaded files
RewriteRule ^([_0-9a-zA-Z-]+/)?files/(.+) wp-includes/ms-
files.php?file=$2 [L]

# add a trailing slash to /wp-admin
RewriteRule ^([_0-9a-zA-Z-]+/)?wp-admin$ $1wp-admin/
[R=301,L]

RewriteCond %{REQUEST_FILENAME} -f [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^ - [L]
RewriteRule ^([_0-9a-zA-Z-]+/)?(wp-
(content|admin|includes).*) $2 [L]
RewriteRule ^([_0-9a-zA-Z-]+/)?(.*\.\php)$ $2 [L]
RewriteRule . index.php [L]
```

If you are using Nginx, please review the following page:

wordpress.org/support/article/nginx

After making the required changes, you may be required to log back into WordPress. Multisite is now enabled and installed and ready to use!

MULTISITE FUNCTIONS

When Multisite is enabled, an entirely new set of features and functions become available for plugin developers to take advantage of. Understanding what functions are available can help you include Multisite-specific functionality in the plugins you create. It can also help to understand how you can make your plugins Multisite-compatible from the beginning.

The Site ID

Every site has a unique ID. This ID will be used in just about every Multisite-specific function you use and is how WordPress determines what site you want to work with. The ID is also used in the prefix of the database tables for each respective site.

For example, if you enable Multisite and create a second site in your network, WordPress creates several database tables prefixed like `wp_2_posts`, where `wp_` is the table prefix you defined when installing WordPress, and `2_` is the blog ID of the new site. As you create additional sites, WordPress creates additional database tables in the same manner.

The blog ID is stored in the global variable `$blog_id`, as shown here:

```
<?php
global $blog_id;
echo 'Current blog ID: ' . $blog_id;
?>
```

The `$blog_id` global variable does exist in standard WordPress but will always be 1. In Multisite mode, the blog ID will be the ID of the blog the current user is viewing.

Common Functions

When working with WordPress Multisite, you can take advantage of some common functions. The first function is called `is_multisite()` and

determines whether Multisite support is enabled. Look at the following example:

```
<?php
if ( is_multisite() ) {
    echo 'Multisite is enabled';
}
?>
```

As you can see, this function does not accept any parameters. It simply checks whether Multisite is enabled in WordPress and returns `True` if so and `False` if not. Anytime you plan on using Multisite-specific functions in WordPress, it's imperative that you use this function to verify that Multisite is running. If Multisite is not running, the default Multisite functions will not be available for use in WordPress, resulting in fatal errors in your plugin.

Another useful function for retrieving network site posts is `get_blog_post()`. This function retrieves a post from any site in the network and is also a good example of a function you can study to better understand how switching between sites works.

```
<?php get_blog_post( $blog_id, $post_id ); ?>
```

The function accepts two parameters: `$site_id` and `$post_id`.

```
<?php
//set blog and post ID
$multisite_blog_id = 2;
$multisite_post_id = 1;

//load the post data
$post_details = get_blog_post(
    $multisite_blog_id, $multisite_post_id );

//display the post title and content
var_dump( $post_details );
?>
```

This example assumes you have a site with an ID of 2 and you want to retrieve post ID 1. This is a quick-and-easy way to retrieve a post from any site in your network.

It can also be useful to retrieve specific information about a site you are working with. To retrieve site information, you can use the `get_blog_details()` function.

```
<?php get_blog_details( $fields, $get_all ); ?>
```

The function accepts two parameters.

- `$fields`: A blog ID, a blog name, or an array of fields to query against
- `$get_all`: Whether to retrieve all details

This function returns an object containing all public variables stored in the `wp_blogs` table. You can also retrieve a single, specific variable.

```
<?php
$blog_details = get_blog_details( 1 );
var_dump( $blog_details );
?>
```

Running the preceding code will produce the following object output:

```
WP_Site Object
(
    [blog_id] => 1
    [site_id] => 1
    [domain] => example.com
    [path] => /
    [registered] => 2010-10-31 19:14:59
    [last_updated] => 2010-11-11 14:19:34
    [public] => 1
    [archived] => 0
    [mature] => 0
    [spam] => 0
    [deleted] => 0
    [lang_id] => 0
    [blogname] => Example Website
    [siteurl] => http://example.com
    [post_count] => 420
)
```

As you can see, there is a lot of valuable data returned about the site specified. You can also retrieve a single option value by stating the name to return, as shown here:

```
<?php
echo 'Total post count: ' . get_blog_details( 1 )-
>post_count;
?>
```

It can also be helpful to retrieve a list of all available sites in your Multisite network. A helpful function to accomplish this is the `get_sites()` function. This function will retrieve a list of sites in your network matching the requested arguments. Let's look at the following example:

```
<?php
//get a list of all public site IDs
$args = array (
    'public' => '1'
);

$sites = get_sites( $args );

if ( is_array( $sites ) ) {

    //loop through the site IDs
    foreach ( $sites as $site ) {

        //display each site name
        echo get_blog_details( $site->blog_id )->blogname .
        '<br/>';

    }
}

?>
```

This is a basic example showing you how to retrieve all network sites that are marked public. The `get_sites()` function accepts a number of arguments, which you can view a full list of at developer.wordpress.org/reference/functions/get_sites.

Switching and Restoring Sites

One major advantage to using Multisite is how easy it is to aggregate content and other data between different sites in your network.

You can use two primary functions to pull data from sites in your network. The first of these functions is `switch_to_blog()`. This function enables you

to switch to any site in your network.

```
<?php switch_to_blog( $site_id ); ?>
```

The function accepts one parameter.

► `$site_id`: The ID of the site you want to switch to

The second function is `restore_current_blog()`. This function restores the user to the current site. You should always execute this function after calling `switch_to_blog()`. If not, everything that processes after the switch will pull from the site you switched to, not the current site. This can mess up your widgets, site settings, and more.

Now look at an example. In this example, you create a custom settings page and display posts from blog ID 3.

```
<?php
add_action( 'admin_menu', 'pdev_multisite_switch_menu' );

function pdev_multisite_switch_menu() {

    //create custom top-level menu
    add_menu_page( 'Multisite Switch', 'Multisite Switch',
        'manage_options', 'pdev-network-switch',
        'pdev_multisite_switch_page' );

}

?>
```

First create a custom top-level menu. This will point to the `pdev_multisite_switch_page()` function, which will be the display page for the posts from site 3.

```
<?php
function pdev_multisite_switch_page() {

    if ( is_multisite() ) {

        //switch to blog ID 3
        switch_to_blog( 3 );

        //create a custom Loop
        $recent_posts = new WP_Query();
        $recent_posts->query( 'posts_per_page=5' );
    }
}
```

```

//start the custom Loop
while ( $recent_posts->have_posts() ) :
    $recent_posts->the_post();

    //store the recent posts in a variable
    echo '<p><a href="' . get_permalink() . '">' .
        get_the_title() . '</a></p> ';

endwhile;

//restore the current site
restore_current_blog();

}

?>

```

As always, you need to verify that Multisite is enabled using the `is_multisite()` function check. If Multisite is not enabled, the `switch_to_blog()` and `restore_current_blog()` functions will not be available to use in your plugin. Next, use the `switch_to_blog()` function to switch to blog ID 3. In this case, you hard-coded the blog ID, but this could always be a dynamic variable set by a user. Now that you've switched to the site you want to pull content from, you need to create a custom loop to retrieve the content.

To create the custom loop, you define a variable named `$recent_posts` and instantiate an instance of `WP_Query`. Next set the query parameters; in this case, you set `posts_per_page` to 5. This returns the five latest posts found. Now that `WP_Query` has been defined, it's time to execute the loop and retrieve the results. You do this with the `have_posts()` and `the_post()` functions. The custom loop will then echo out the posts found from the query.

The final step is to execute `restore_current_blog()`. If you did not run this function, WordPress would stay on the site you switched to. If you execute additional loops below this, they would all pull from blog ID 3 and not from the current site you are viewing.

Now when you visit the plugin settings page, the latest five posts from blog ID 3 display. Review the entire plugin, shown here:

```
<?php
/*
Plugin Name: Multisite Switch Example Plugin
Plugin URI: https://example.com/wordpress-plugins/my-plugin
Description: A plugin to demonstrate Multisite site
switching
Version: 1.0
Author: Brad Williams
Author URI: https://wrox.com
License: GPLv2
*/
add_action( 'admin_menu', 'pdev_multisite_switch_menu' );

function pdev_multisite_switch_menu() {

    //create custom top-level menu
    add_menu_page( 'Multisite Switch', 'Multisite Switch',
        'manage_options',
        'pdev-network-switch', 'pdev_multisite_switch_page'
);
}

function pdev_multisite_switch_page() {

    if ( is_multisite() ) {

        //switch to blog ID 3
        switch_to_blog( 3 );

        //create a custom Loop
        $recent_posts = new WP_Query();
        $recent_posts->query( 'posts_per_page=5' );

        //start the custom Loop
        while ( $recent_posts->have_posts() ) :
            $recent_posts->the_post();

            //store the recent posts in a variable
            echo '<p><a href="' . get_permalink() . '">' .
                get_the_title() . '</a></p>';

        endwhile;

        //restore the current site
        restore_current_blog();
    }
}
```

```
 }  
 }
```

This is a basic example that demonstrates the power of the `switch_to_blog()` functionality in Multisite.

The `switch_to_blog()` function is not just limited to site content. You can also retrieve other WordPress data including widgets, sidebars, menus, and more. Basically, any data stored in the content database tables (`wp_ID_tablename`) is available when using the `switch_to_blog()` function. Now look at a few examples. In the following example, you can assume you have a site with an ID of 3 and you want to retrieve a navigation menu from the site:

```
<?php  
//switch to blog ID 3  
switch_to_blog( 3 );  
  
//display the nav menu Main Menu  
wp_nav_menu( 'Main Menu' );  
  
//restore the current site  
restore_current_blog();  
?>
```

First run `switch_to_blog()` to switch to blog ID 3. Next use the `wp_nav_menu()` function to display a menu named Main Menu from the site. Finally, run the `restore_current_blog()` function to reset to the blog you are viewing. The end result displays the nav menu Main Menu created on site 3 anywhere you run this code in your network.

As another example, you can also easily load a site's sidebar using the same method.

```
<?php  
//switch to blog ID 34  
switch_to_blog( 34 );  
  
//load the primary sidebar  
get_sidebar();  
  
//restore the current site
```

```
restore_current_blog();
?>
```

It's important to note that `switch_to_blog()` is database-only. This means a site's plugins are not included in a switch. So, if site 2 has the Halloween Revenge plugin running and you switch to site 2, Halloween Revenge will not be available for use unless it is also activated on the site performing the switch.

Network Content Shortcode Examples

Now take the switch example and integrate shortcode support. This plugin enables you to add a shortcode to a post on the current site, defining what blog ID you want some other posts from, and display those posts on your current post or page.

First create a new shortcode using the `add_shortcode()` function, which will be introduced in [Chapter 14](#), “The Kitchen Sink.”

```
<?php
add_shortcode( 'network_posts',
'pdev_multisite_network_posts' );
?>
```

The new shortcode will be `[network_posts]`. Next create the function to generate the network posts to display when the shortcode is used in a post or page.

```
<?php
function pdev_multisite_network_posts( $attr ) {
    extract( shortcode_atts( array(
        "site_id"      =>      '1',
        "num"          =>      '5'
    ), $attr ) );

    if ( is_multisite() ) {

        $return_posts = '';

        //switch to site set in the shortcode
        switch_to_blog( absint( $site_id ) );

        //create a custom Loop
        $recent_posts = new WP_Query();
```

```

        $recent_posts->query( 'posts_per_page=' . absint(
$num ) );

        //start the custom Loop
        while ( $recent_posts->have_posts() ) :
            $recent_posts->the_post();

            //store the recent posts in a variable
            $title = get_the_title();
            $permalink = get_the_permalink();
            $return_posts .= '<p><a href="' . $permalink '">' .
$title . '</a></p>';

        endwhile;

        //restore the current site
        restore_current_blog();

        //return the results to display
        return $return_posts;

    }
}
?>

```

The shortcode can accept two parameters: `site_id` and `num`. This enables the user to set which site in the network to pull the posts from and how many to display. As always, check to verify that Multisite is enabled on the site before proceeding.

`$return_posts` is the variable that stores all the posts to return to the shortcode for display, so start by setting that variable to nothing to flush it out. Next use the `switch_to_blog()` function to switch to the site specified in the shortcode. If the user did not set a specific blog ID, it will default to 1.

Now it's time to create a custom loop to retrieve the posts to display. You can see that the `posts_per_page` parameter is set to `$num`, which is set in the shortcode. If the user does not set the number of posts to display, it defaults to 5. Next loop through the posts loaded and store them in `$return_posts`.

After the custom loop finishes running, you need to execute `restore_current_blog()`. This resets the site to the site you are viewing,

not the site you switched to earlier. The final step is to return `$return_posts`. This replaces the shortcode in a post or page with the custom loop results.

Now you can easily retrieve posts from any site in your network using the shortcode such as `[network_posts site_id="3" num="10"]`. Review the full plugin, shown here:

```
<?php
/*
Plugin Name: Multisite Switch Shortcode Plugin
Plugin URI: https://example.com/wordpress-plugins/my-plugin
Description: A plugin for aggregating content using a
shortcode
Version: 1.0
Author: Brad Williams
Author URI: https://wrox.com
License: GPLv2
*/
add_shortcode( 'network_posts',
'pdev_multisite_network_posts' );

function pdev_multisite_network_posts( $attr ) {
    extract( shortcode_atts( array(
        'site_id'      =>      '1',
        'num'          =>      '5'
    ), $attr ) );
    if ( is_multisite() ) {
        $return_posts = '';
        //switch to site set in the shortcode
        switch_to_blog( absint( $site_id ) );
        //create a custom Loop
        $recent_posts = new WP_Query();
        $recent_posts->query( 'posts_per_page=' . absint(
        $num ) );
        //start the custom Loop
        while ( $recent_posts->have_posts() ) :
            $recent_posts->the_post();
        //store the recent posts in a variable
```

```

        $return_posts .= '<p><a href="'
.get_permalink().
'">' .get_the_title() . '</a></p>';

endwhile;

//restore the current site
restore_current_blog();

//return the results to display
return $return_posts;

}
}

```

Now take the switch shortcode example to the next level and retrieve posts from multiple sites in the network and display them based on the latest post date. As in the previous example, use the `add_shortcode()` function to register the shortcode in your plugin.

```

<?php
add_shortcode( 'latest_network_posts',
    'pdev_multisite_latest_network_posts' );
?>

```

Next create your custom `pdev_multisite_latest_network_posts()` function.

```

<?php
function pdev_multisite_latest_network_posts() {

    if ( is_multisite() ) {

        $return_posts = '';

```

As always, check to verify that Multisite is enabled using the `is_multisite()` function. You can also set `$return_posts` to nothing to flush it out. Now it's time to retrieve the posts.

```

//get posts from current site
$local_posts = get_posts( 'numberposts=5' );

//switch to blog ID 3
switch_to_blog( 3 );

//get posts from another site

```

```
$network_posts = get_posts( 'numberposts=5' );  
  
//restore the current site  
restore_current_blog();
```

Use the `get_posts()` function to retrieve the latest five posts from the current site. Next switch to blog ID 3 and run the same `get_posts()` function to retrieve the five latest posts from that site. Notice that you are storing the returned array values in separate variables: `$local_posts` and `$network_posts`. Finally, call `restore_current_blog()` to reset to the current site you are on.

Now that you have five posts from each site stored in separate arrays, you need to merge them into a single array.

```
//merge the two arrays  
$posts = array_merge( $local_posts, $network_posts );
```

Now that you have a single array of posts, you need to sort the posts based on the post date so that they are in proper reverse chronological order with the latest post first. Use the PHP `usort()` function to sort the array based on a custom comparison function you will create later.

```
//sort the post results by date  
usort( $posts, 'pdev_multisite_sort_posts_array' );
```

Now that the posts are in the proper order in the array, you need to loop through the results and assign them to the `$return_posts` variable.

```
foreach ( $posts as $post ) {  
  
    //store latest posts in a variable  
    $return_posts .= $post->post_title . ' - posted on '  
    . $post->post_date . '<br/>';  
  
}
```

Use a standard `foreach` PHP loop to loop through the results. Finally, return the results for display by the shortcode.

```
//return the results to display  
return $return_posts;
```

```
    }
}
```

The final step is to create the custom function `pdev_multisite_sort_posts_array()` to sort the post array by the date that was called earlier from the `usort` function.

```
//sort the array by date
function(pdev_multisite_sort_posts_array( $a, $b ) {

    //if dates are the same return 0
    if ($a->post_date == $b->post_date)
        return 0;

    //ternary operator to determine which date is newer
    return $a->post_date < $b->post_date ? 1 : -1;

}
```

This function simply compares two values and returns either a 1 or -1 based on which is greater. The `usort()` function sorts based on the number assigned.

Now the shortcode `[latest_network_posts]` will display the latest 10 posts between two sites based on the publish date. Review the entire plugin code, shown here:

```
<?php
/*
Plugin Name: Multisite Latest Network Posts Plugin
Plugin URI: https://example.com/wordpress-plugins/my-plugin
Description: Displays the latest posts from multiple sites
Version: 1.0
Author: Brad Williams
Author URI: https://wrox.com
License: GPLv2
*/
add_shortcode( 'latest_network_posts',
    'pdev_multisite_latest_network_posts' );
function(pdev_multisite_latest_network_posts() {
    if ( is_multisite() ) {
        $return_posts = '';
    }
}
```

```

//get posts from current site
$local_posts = get_posts( 'numberposts=5' );

//switch to blog ID 3
switch_to_blog( 3 );

//get posts from another site
$network_posts = get_posts( 'numberposts=5' );

//restore the current site
restore_current_blog();

//merge the two arrays
$posts = array_merge( $local_posts, $network_posts
);

//sort the post results by date
usort( $posts, 'pdev_multisite_sort_posts_array' );

foreach ( $posts as $post ) {

    //store latest posts in a variable
    $return_posts .= $post->post_title . ' - posted
on '
        . $post->post_date . '<br/>';

}

//return the results to display
return $return_posts;

}

//sort the array by date
function pdev_multisite_sort_posts_array( $a, $b ) {

    //if dates are the same return 0
    if ( $a->post_date == $b->post_date )
        return 0;

    //ternary operator to determine which date is newer
    return $a->post_date < $b->post_date ? 1 : -1;

}

```

A Network Content Widget Example

Another common task when working in a Multisite environment is a widget to display recent posts from sites in the network. You can create a plugin with a widget to display the recent posts from any site in the network.

```
<?php
//widgets_init action hook to execute custom function
add_action( 'widgets_init', 'pdev_multisite_register_widget'
);

//register our widget
function pdev_multisite_register_widget() {
    register_widget( 'pdev_multisite_widget' );
}
?>
```

First use the `widgets_init` action hook to run the custom function to register your new widget. In this example, the new widget will be registered as `pdev_multisite_widget`. Next create a new class using the registered widget name and extending the `WP_Widget`.

```
//pdev_multisite_widget class
class PDEV_Multisite_Widget extends WP_Widget {

    //process our new widget
    function __construct() {

        $widget_ops = array( 'classname' =>
'pdev_multisite_widget',
        'description' =>
            'Display recent posts from a network site.'
);
        parent::__construct( 'pdev_multisite_widget_posts',
'Multisite Recent Posts', $widget_ops );

    }
}
```

You also define the widget settings. You set the widget name to Multisite Recent Posts, the description of what the widget does, and the custom class name that will be used when displaying the widget.

Now it's time to create the widget settings form. This widget contains three settings: the title, the site to load recent posts from, and the number of posts to display.

```

//build our widget settings form
function form( $instance ) {
    global $wpdb;

    $defaults = array( 'title' => 'Recent Posts',
                      'disp_number' => '5' );
    $instance = wp_parse_args( (array) $instance,
    $defaults );
    $title = $instance['title'];
    $siteid = $instance['siteid'];
    $disp_number = $instance['disp_number'];

```

You will be making a custom database query to retrieve the network blog IDs, so you need to define \$wpdb as a global variable. The widget defaults are set in the \$defaults variable; in this case, you set the default title to Recent Posts and the default number of posts to display to 5. Next, the instance values are loaded, which are your widget setting values.

Now that you have loaded the widget values, you need to add the form field settings for the widget. The first field is a text field to store the widget title that will be displayed.

```

//title textfield widget option
echo '<p>Title: <input class="widefat" name=""'
    . $this->get_field_name( 'title' )
    .'" type="text" value=""'
    . esc_attr( $title ). '"></p>';

```

As always, you want to use the proper escaping function when displaying data entered by a user, in this case esc_attr() to display the \$title value.

The next field to add is a select form to set which site in the network to display recent posts from. To create this form field, you need to retrieve a list of all public blog IDs in your network. You'll use the get_sites() function to retrieve the IDs.

```

//get a list of all public site IDs
$args = array (
    'public' => '1'
);

$sites = get_sites( $args );

```

The function retrieves all public blog IDs in your Multisite network and returns them in an array stored in the \$sites variable. Now that you have

the blog IDs, you need to loop through the results to build the select list.

```
if ( is_array( $sites ) ) {

    echo '<p>';
    echo 'Site to display recent posts';
    echo '<select name="' . $this->get_field_name(
'siteid' )
        . '" class="widefat">';

    //loop through the blog IDs
    foreach ( $sites as $site ) {

        //display each site as an option
        echo '<option value="' . $site->blog_id. '" '
            . selected( $site->blog_id, $siteid )
            . '>' . get_blog_details( $site->blog_id
)->blogname
            . '</option>';

    }

    echo '</select>';
    echo '</p>';
}
```

Before working with an array, it's a good practice to verify that it is actually an array. You can do so using the PHP function `is_array()`. After you confirm that `$blogs` is an array, you can display the option text and select field. To display each site as an option, loop through the array values. Use the `get_blog_details()` function to display the site name in the option field. The `$blog` variable, which stores the blog ID, is set to the value of the option field.

The final form field to display is the number of posts option.

```
//number to display textfield widget option
echo '<p>Number to display: <input class="widefat"
name="" '
        . $this->get_field_name( 'disp_number' ). ''
type="text" '
        value="" . esc_attr( $disp_number ). '"></p>';

}
```

Just like the title option before, this is a standard text form field to store the number of posts to display. That's the final widget form field, so be sure to close out the function with }.

Next you need to save your widget settings using the update widget class function.

```
//save the widget settings
function update( $new_instance, $old_instance ) {

    $instance = $old_instance;
    $instance['title'] = sanitize_text_field(
$new_instance['title'] );
    $instance['siteid'] = absint(
$new_instance['siteid'] );
    $instance['disp_number'] =
        absint( $new_instance['disp_number'] );

    return $instance;
}
```

The widget class will handle saving the options for you. Be sure to sanitize the widget settings. Both siteid and disp_number should always be a number, so use the absint() function to verify that the setting is a positive integer.

The final step is to display the widget.

```
//display the widget
function widget( $args, $instance ) {
    extract( $args );

    echo $before_widget;

    //load the widget options
    $title = apply_filters( 'widget_title',
$instance['title'] );
    $siteid = empty( $instance['siteid'] ) ? 1 :
        $instance['siteid'];
    $disp_number = empty( $instance['disp_number'] ) ? 5
    :
        $instance['disp_number'];

    //display the widget title
    if ( !empty( $title ) ) {
        echo $before_title . esc_html( $title ) .
```

```

$after_title;
}

echo '<ul>';

```

First, extract the \$args variable to gain access to the global theme values like \$before_widget and \$after_widget. Next load the widget settings. The \$siteid and \$disp_number variables are both using a ternary operator to set their values. This means if the option value is empty, it will be set to a default value. \$siteid would default to 1, and \$disp_number would default to 5.

Now display the \$title, surrounded by the \$before_title and \$after_title global theme values. Now it's time to display the recent posts from the site saved in the widget.

```

//switch to site saved
switch_to_blog( absint( $siteid ) );

//create a custom loop
$recent_posts = new WP_Query();
$recent_posts->query( 'posts_per_page='
. absint( $disp_number ) );

//start the custom loop
while ( $recent_posts->have_posts() ) :
$recent_posts->the_post();

//display the recent post title with link
echo '<li><a href="' . get_permalink() . '">'
. get_the_title() . '</a></li>';

endwhile;

//restore the current site
restore_current_blog();

echo '</ul>';
echo $after_widget;

}

```

Using the `switch_to_blog()` function again, the widget switches to the site saved in the widget settings. After the site has been loaded, create a custom loop using the `WP_Query` class. The `posts_per_page` query parameter is defined by the `$disp_number` widget setting. The recent posts display in an unordered list using a `while` loop. After the loop completes, you need to restore the current site using `restore_current_blog()`.

You now have a Multisite widget to easily display posts from any site in your network! This simple example shows the power of aggregating content throughout a Multisite network in WordPress and how easy it is to accomplish that.

```
<?php
/*
Plugin Name: Multisite Recent Posts Widget
Plugin URI: https://example.com
Description: Retrieves the most recent posts in a Multisite
network
Author: Brad Williams
Version: 1.0
Author URI: https://wrox.com
*/
//widgets_init action hook to execute custom function
add_action( 'widgets_init', 'pdev_multisite_register_widget' );
//register our widget
function pdev_multisite_register_widget() {
    register_widget( 'pdev_multisite_widget' );
}
//pdev_multisite_widget class
class PDEV_Multisite_Widget extends WP_Widget {
    //process our new widget
    function __construct() {
        $widget_ops = array( 'classname' =>
'pdev_multisite_widget',
        'description' =>
            'Display recent posts from a network site.'
);
        parent::__construct( 'pdev_multisite_widget_posts',
'Multisite Recent Posts', $widget_ops );
    }
}
```

```

}

//build our widget settings form
function form( $instance ) {
    global $wpdb;

    $defaults = array( 'title' => 'Recent Posts',
                      'disp_number' => '5' );
    $instance = wp_parse_args( (array) $instance,
$defaults );
    $title = $instance['title'];
    $siteid = $instance['siteid'];
    $disp_number = $instance['disp_number'];

    //title textfield widget option
    echo '<p>Title: <input class="widefat" name=""'
        . $this->get_field_name( 'title' )
        . '" type="text" value="' . esc_attr( $title )
        . '"/></p>';

    //get a list of all public site IDs
    $args = array (
        'public' => '1'
    );

    $sites = get_sites( $args );

    if ( is_array( $sites ) ) {

        echo '<p>';
        echo 'Site to display recent posts';
        echo '<select name="' . $this-
>get_field_name('siteid')
        . '" class="widefat">';

        //loop through the blog IDs
        foreach ($sites as $site) {

            //display each site as an option
            echo '<option value="' . $site->blog_id. '"'
                . selected( $site->blog_id, $siteid )
                . '>' . get_blog_details( $site->blog_id
)->blogname
                . '</option>';

        }
    }
}

```

```

        echo '</select>';
        echo '</p>';
    }

    //number to display textfield widget option
    echo '<p>Number to display: <input class="widefat"
name=""'
        . $this->get_field_name( 'disp_number' ). ''
type="text"
        value="" . esc_attr( $disp_number ). '"></p>';

}

//save the widget settings
function update( $new_instance, $old_instance ) {

    $instance = $old_instance;
    $instance['title'] = strip_tags(
$new_instance['title'] );
    $instance['siteid'] = absint(
$new_instance['siteid'] );
    $instance['disp_number'] =
        absint( $new_instance['disp_number'] );

    return $instance;
}

//display the widget
function widget( $args, $instance ) {
    extract( $args );

    echo $before_widget;

    //load the widget options
    $title = apply_filters( 'widget_title',
$instance['title'] );
    $siteid = empty( $instance['siteid'] ) ? 1 :
        $instance['siteid'];
    $disp_number = empty( $instance['disp_number'] ) ?
5 :
        $instance['disp_number'];

    //display the widget title
    if ( !empty( $title ) ) { echo $before_title .
$title
        . $after_title; };

    echo '<ul>';

```

```

//switch to site saved
switch_to_blog( absint( $siteid ) );

//create a custom loop
$recent_posts = new WP_Query();
$recent_posts->query( 'posts_per_page='
    .absint( $disp_number ) );

//start the custom Loop
while ( $recent_posts->have_posts() ) :
    $recent_posts->the_post();

    //display the recent post title with link
    echo '<li><a href="' . get_permalink() . '">' .
        get_the_title() . '</a></li>';

endwhile;

//restore the current site
restore_current_blog();

echo '</ul>';
echo $after_widget;

}

}

```

Creating a New Site

You can easily create new sites in your Multisite network in the Dashboard of WordPress. But what if you want to create a new site in your plugin? As always, there's a function for that, and it's called `wpmu_create_blog()`.

```
<?php wpmu_create_blog( $domain, $path, $title,
    $user_id, $meta, $site_id); ?>
```

This function accepts six parameters.

- `$domain`: The domain of the new site.
- `$path`: The path of the new site. This is the subdirectory or subdomain name depending on which setup you use.
- `$title`: The title of the new site.

- `$user_id`: The user ID of the user account who will be the site admin.
- `$meta`: Additional meta information.
- `$site_id`: The blog ID of the site to be created.

The only required parameters are the first four; the last two are optional. If the new site is created successfully, the function returns the newly created blog ID. This function does all the heavy lifting of creating database tables, adding default content, assigning any user roles, and so on.

As you probably noticed, the function begins with `wpmu_`. Many of the Multisite functions were once a part of WordPress MU, prior to the merging of the two code bases in WordPress 3.0. These function names can contain `wpmu`, or `blog`, to support backward compatibility.

As an example, create a plugin that enables users to create sites in WordPress Multisite. First, create a custom top-level menu for the plugin page.

```
<?php
add_action( 'admin_menu', 'pdev_multisite_create_menu' );

function pdev_multisite_create_menu() {

    //create custom top-level menu
    add_menu_page( 'Multisite Create Site Page',
        'Multisite Create Site',
        'manage_options', 'pdev-network-create',
        'pdev_multisite_create_sites_page' );

}

?>
```

Now create the function to display a form for creating a new site.

```
<?php
function pdev_multisite_create_sites_page() {

    //check if multisite is enabled
    if ( is_multisite() ) {
```

As always, you need to verify that Multisite is enabled before using any Multisite-specific functions. Next add the code to retrieve the form fields submitted and create a new site in the network with the values.

```

<?php
//if the form was submitted lets process it
if ( isset( $_POST['create_site'] ) ) {

    //populate the variables based on form values
    $domain = sanitize_text_field( $_POST['domain'] )
};

    $path = sanitize_text_field( $_POST['path'] );
    $title = sanitize_text_field( $_POST['title'] );
    $user_id = absint( $_POST['user_id'] );

    //verify the required values are set
    if ( $domain && $path && $title && $user_id ) {

        //create the new site in WordPress
        $new_site = wpmu_create_blog( $domain,
$path,
            $title, $user_id );

        //if successfully display a message
        if ( $new_site ) {

            echo '<div class="notice notice-success
is-dismissible">New site '
                . $new_site. ' created successfully!
</div>';

        }

        //if required values are not set display an
        error
    } else {

        echo '<div class="notice notice-error is-
-dismissible">
            New site could not be created.
            Required fields are missing</div>';

    }

}

?>

```

First check whether `$_POST['create_site']` is set. This will be set only if the form has been submitted. Next populate the variables based on the form

entries. Notice you'll be using the proper escaping functions to verify that the data submitted from the form is escaped properly.

Next verify that \$domain, \$path, \$title, and \$user_id all have values because they are the required fields when creating sites using `wpmu_create_blog()`. If the values are not filled out, an error message displays. After you verify that all values exist, it's time to execute the `wpmu_create_blog()` function to create the new site. If the site is created successfully, the variable \$new_site will contain the ID of the newly created site, and a success message will be displayed.

The final piece is to create the form for the new site fields.

```
<div class="wrap">
    <h2>Create New Site</h2>
    <form method="post">
        <table class="form-table">
            <tr valign="top">
                <th scope="row">
                    <label for="fname">Domain</label>
                </th>
                <td><input maxlength="45" size="25"
name="domain"
value="<?php echo DOMAIN_CURRENT_SITE; ?>" />
                    </td>
                </tr>
                <tr valign="top">
                    <th scope="row"><label
for="fname">Path</label></th>
                    <td>
                        <input maxlength="45" size="10"
name="path"/>
                    </td>
                </tr>
                <tr valign="top">
                    <th scope="row">
                        <label for="fname">Title</label>
                    </th>
                    <td>
                        <input maxlength="45" size="25"
name="title"/>
                    </td>
                </tr>
                <tr valign="top">
                    <th scope="row">
```

```

                <label for="fname">User ID</label>
            </th>
            <td>
                <input maxlength="45" size="3"
name="user_id"/>
            </td>
        </tr>
        <tr valign="top">
            <td>
                <input type="submit" name="create_site"
value="Create Site" class="button-
primary"/>
            <input type="submit" name="reset"
value="Reset" class="button-secondary"/>
            </td>
        </tr>
    </table>
</form>
</div>

```

This is a fairly basic form that accepts the parameters required to create a new site.

```

<?php
/*
Plugin Name: Multisite Create Site Example Plugin
Plugin URI: http://example.com/wordpress-plugins/my-plugin
Description: A plugin to demonstrate creating sites in
Multisite
Version: 1.0
Author: Brad Williams
Author URI: http://wrox.com
License: GPLv2
*/

```

```

add_action( 'admin_menu', 'pdev_multisite_create_menu' );

function pdev_multisite_create_menu() {

    //create custom top-level menu
    add_menu_page( 'Multisite Create Site Page',
        'Multisite Create Site',
        'manage_options', 'pdev-network-create',
        'pdev_multisite_create_site_settings' );

}

```

```

function pdev_multisite_create_site_settings() {

    //check if multisite is enabled
    if ( is_multisite() ) {

        //if the form was submitted lets process it
        if ( isset( $_POST['create_site'] ) ) {

            //populate the variables based on form values
            $domain = sanitize_text_field( $_POST['domain'] )
        };

        $path = sanitize_text_field( $_POST['path'] );
        $title = sanitize_text_field( $_POST['title'] );
        $user_id = absint( $_POST['user_id'] );

        //verify the required values are set
        if ( $domain && $path && $title && $user_id ) {

            //create the new site in WordPress
            $new_site = wpmu_create_blog( $domain,
$path,
                $title, $user_id );

            //if successfully display a message
            if ( $new_site ) {

                echo '<div class="notice notice-success
is-dismissible">New site '
                    . $new_site. ' created successfully!
</div>';

            }

            //if required values are not set display an
            error
        } else {

            echo '<div class="notice notice-error is-
dismissible">
                New site could not be created.
                Required fields are missing
            </div>';

        }

    }

}

```

```

?>
<div class="wrap">
    <h2>Create New Site</h2>
    <form method="post">
        <table class="form-table">
            <tr valign="top">
                <th scope="row">
                    <label for="fname">Domain</label>
                </th>
                <td><input maxlength="45" size="25"
name="domain"
                                value="<?php echo
DOMAIN_CURRENT_SITE; ?>"/>
                </td>
            </tr>
            <tr valign="top">
                <th scope="row"><label
for="fname">Path</label></th>
                <td>
                    <input maxlength="45" size="10"
name="path"/>
                </td>
            </tr>
            <tr valign="top">
                <th scope="row"><label
for="fname">Title</label></th>
                <td>
                    <input maxlength="45" size="25"
name="title"/>
                </td>
            </tr>
            <tr valign="top">
                <th scope="row">
                    <label for="fname">User ID</label>
                </th>
                <td>
                    <input maxlength="45" size="3"
name="user_id"/>
                </td>
            </tr>
            <tr valign="top">
                <td>
                    <input type="submit" name="create_site"
value="Create Site" class="button-
primary"/>
                    <input type="submit" name="reset"
value="Reset"
                                class="button-secondary"/>
                </td>
            </tr>
        </table>
    </form>
</div>

```

```

        </td>
    </tr>
    </table>
    </form>
</div>
<?php
} else {
    echo '<p>Multisite is not enabled</p>';
}
}

```

There is also an easy method to update a site's status. This is useful if you want to dynamically archive a site or flag a site as spam. To do so, use the `update_blog_status()` function.

```
<?php update_blog_status( $blog_id, $pref, $value, $refresh ); ?>
```

The function accepts four parameters.

- `$site_id`: The blog ID of the site to update
- `$pref`: The status type to update
- `$value`: The new value of the status
- `$refresh`: Whether to refresh the site details cache

The first three parameters are required. The `$pref` parameter is the status to update, which accepts `site_id`, `domain`, `path`, `registered`, `last_updated`, `public`, `archived`, `mature`, `spam`, `deleted`, and `lang_id`. In this example, you update a site in your network to be archived.

```
<?php
update_blog_status( $blog_id, 'archived', 1 );
?>
```

Site Options

Site options in Multisite are stored identically to non-Multisite options, but we recommend you use the dedicated family of functions to be explicit about what you are trying to do and why.

- `add_blog_option()`: Creates a new option
- `update_blog_option()`: Updates an option and creates it if it doesn't exist
- `get_blog_option()`: Loads a site option that already exists
- `delete_blog_option()`: Deletes a site option

The major difference between this set of functions and the standard option functions is you have to pass a blog ID parameter to each function. The function will then switch to the site specified, handle the option task, and then switch back to the current site.

```
<?php add_blog_option( $blog_id, $key, $value ); ?>
```

The `$site_id` value is the ID of the site you want to add an option to. The `$key` value is the name of the option to add, and `$value` is the value of the new option.

Loading site options is just as easy. Using the `get_blog_option()` function, you can load any site-specific option required.

```
<?php
$site_id= 3;
echo '<p>Site ID: '.$site_id.'</p>';
echo '<p>Site Name: ' .get_blog_option( $blog_id, 'blogname' )
)
. '</p>';
echo '<p>Site URL: ' .get_blog_option( $blog_id, 'siteurl' )
. '</p>';
?>
```

Network Options

Network options in Multisite are like Site Options but relate specifically to the network itself and are stored more like metadata than options.

- `add_network_option()`: Creates a new option
- `update_network_option()`: Updates an option and creates it if it doesn't exist
- `get_network_option()`: Loads an option that already exists

- `delete_network_option()`: Deletes an option

All of these will require you to either pass in a network ID as the first parameter or pass in `null` to use the current network. The function will then switch to the site specified, handle the option task, and then switch back to the current site.

```
<?php add_network_option( $network_id, $key, $value ); ?>
```

The `$network_id` value is the ID of the network you want to add an option to. The `$key` value is the name of the option to add, and `$value` is the value of the new option.

Loading network options is just as easy. Using the `get_network_option()` function, you can load any network-specific option required.

```
<?php
$network_id= 3;
echo '<p>Network ID: ' . $network_id . '</p>';
echo '<p>Network Name: ' . get_network_option( $network_id,
'sitename' )
. '</p>';
echo '<p>Network URL: ' . get_network_option( $network_id,
'siteurl' ) . '</p>';
?>
```

Site Meta

Site meta in Multisite are stored globally and are a super useful way to add data to sites in a way that can be queried across multiple sites, whereas Network Options can be per site only, requiring site switching to aggregate them.

- `add_site_meta()`: Creates new metadata
- `update_site_meta()`: Updates metadata and creates it if it doesn't exist
- `get_site_meta()`: Loads metadata that already exists
- `delete_site_meta()`: Deletes metadata

These functions use the Metadata API inside WordPress to make querying for and caching these arbitrary values an absolute breeze.

```
<?php add_site_meta( $site_id, $key, $value ); ?>
```

The `$site_id` value is the ID of the site you want to add metadata to. The `$key` value is the name of the metadata, and `$value` is the value of the new data.

Loading site metadata is just as easy. Using the `get_site_Meta()` function, you can load any site-specific metadata required. WordPress currently does not store any data in this table on its own. It exists simply as a way to extend sites at a global level.

Users and Roles

Users work slightly differently than in standard WordPress. If Allow New Registrations is enabled under Network Settings, visitors to your site can register new accounts. The major difference is that each site in your network will have its own different set of users based on who has access to those sites. Users can also have different roles on different sites throughout the network. Users are not automatically members of every site in your network.

Before executing any code that is site-specific, you should verify that the user logged in is a member of that site. Multisite features multiple functions for working with users. To verify that a user is a member of the site, use the `is_user_member_of_blog()` function.

```
<?php is_user_member_of_blog( 0, $site_id ) ?>
```

The function accepts two parameters, `$user_id` and `$site_id`, which are both optional. If the parameters are not specified, it defaults to the currently logged-in user and the current site you are on.

```
<?php
if ( is_user_member_of_blog() ) {
    //user is a member of this site
}
?>
```

You may specify a user ID if you want to verify that a user is a member of this site.

```
<?php
if ( is_user_member_of_blog( 42 ) ) {
    //user 42 is a member of this site
}
?>
```

You can also specify a blog ID if you want to verify that the user is a member of a specific site.

```
<?php
if ( is_user_member_of_blog( 42, 3 ) ) {
    //user 42 is a member of site ID 3
}
?>
```

Now that you understand how to check whether a user is a member of a site, let's look at how to add members to a site. In Multisite, you use the `add_user_to_blog()` function to add any user in WordPress to a specific site in your network.

```
<?php add_user_to_blog( $blog_id, $user_id, $role ); ?>
```

This function accepts three parameters.

- `$site_id`: The ID of the site you want to add the user to
- `$user_id`: The ID of the user to add
- `$role`: The user role the user will have on the site

Look at the following working example:

```
<?php
$site_id = 18;
$user_id = 4;
$role = 'editor';

add_user_to_blog( $blog_id, $user_id, $role );
?>
```

Now build a real-world example plugin. This plugin auto-adds logged-in users to any site they visit in your network. This is useful if you want users

to become members on every site in your network without manually adding them to each one.

Start by using the `init` action hook to execute your custom function to add users to a site.

```
<?php
add_action( 'init', 'pdev_multisite_add_user_to_site' );
?>
```

Next create the `pdev_multisite_add_user_to_site()` function to add the users.

```
<?php
function pdev_multisite_add_user_to_site() {

    //verify user is logged in before proceeding
    if( !is_user_logged_in() ) {
        return false;
    }

    //verify user is not a member of this site
    if( !is_user_member_of_blog() ) {

        //add user to this site as a subscriber
        $site_id = get_current_blog_id();
        $user_id = get_current_user_id();
        add_user_to_blog( $site_id, $user_id, 'subscriber'
    );

    }
}

?>
```

The first step is to verify that the users are logged in and, if not, return `false` and exit the function. Next confirm if the users are already members of the site they are viewing. If the users are already members, there is no reason to add them again.

The final step is to add the users to the site using the `add_user_to_blog()` function. You'll pass in the blog ID, the current user ID, and the role the users are assigned on the site, in this case `subscriber`. That's it! For this plugin to automatically work across your entire network, you'll need to either upload it to the `/mu-plugins` directory or activate the plugin on the

Network Admin \Rightarrow Plugins page. That forces the plugin to run across all sites in your network.

```
<?php
/*
Plugin Name: Multisite Auto-Add User to Site
Plugin URI: https://example.com/wordpress-plugins/my-plugin
Description: Plugin automatically adds the user to any site
they visit
Version: 1.0
Author: Brad Williams
Author URI: https://wrox.com
License: GPLv2
*/
add_action( 'init', 'pdev_multisite_add_user_to_site' );

function pdev_multisite_add_user_to_site() {

    //verify user is logged in before proceeding
    if( !is_user_logged_in() ) {
        return false;
    }

    //load current blog ID and user data
    global $current_user, $blog_id;
    //verify user is not a member of this site
    if( !is_user_member_of_blog() ) {

        //add user to this site as a subscriber
        add_user_to_blog( $blog_id, $current_user->ID,
        'subscriber' );
    }
}
```

As easily as you can add users, you can also remove users from a site using the `remove_user_from_blog()` function.

```
<?php remove_user_from_blog( $user_id, $blog_id, $reassign
); ?>
```

This function accepts three parameters.

- `$user_id`: The user ID you want to remove

- `$site_id`: The blog ID to remove the user from
- `$reassign`: The user ID to reassign posts to

Look at the following working example:

```
<?php
$user_id = 4;
$site_id = 18;
$reassign = 1;

remove_user_from_blog( $user_id, $blog_id, $reassign );
?>
```

NOTE *Remember, adding and removing users from a site in Multisite is not actually creating or deleting the user in WordPress but instead adding or removing them as a member of that site.*

Another useful function when working with Multisite users is `get_blogs_of_user()`. This function retrieves site information for all sites the specified users are a member of.

```
<?php
$user_id = 1;
$user_sites = get_blogs_of_user( $user_id );
print_r( $user_sites );
?>
```

Running this code example will result in an object array being returned, as shown here:

```
Array
(
    [1] => stdClass Object
        (
            [userblog_id] => 1
            [blogname] => Main Site
            [domain] => example.com
            [path] => /
            [site_id] => 1
            [siteurl] => http://example.com
        )
    [2] => stdClass Object
)
```

```

(
    [userblog_id] => 2
    [blogname] => Halloween Revenge
    [domain] => example.com
    [path] => /myers/
    [site_id] => 1
    [siteurl] => http://example.com/myers
)
[8] => stdClass Object
(
    [userblog_id] => 8
    [blogname] => Freddy Lives
    [domain] => example.com
    [path] => /kruger/
    [site_id] => 1
    [siteurl] => http://example.com/kruger
)
)

```

You can also do a `foreach` loop to display specific data from the array.

```

<?php
$user_id = 1;
$user_sites = get_blogs_of_user( $user_id );

foreach ( $user_sites as $user_site ) {

    echo '<p>' . $user_site->siteurl . '</p>';

}
?>

```

Super Admin

Multisite offers a pseudo-user role: Super Administrator. Super admins have access to the administration areas of the entire WordPress installation, including one specific to the network. The network admin is where all network settings, themes, plugins, and so on, are managed. Super admins also have full control over every site in the network, whereas a regular admin can administer only their specific sites.

In Multisite, you can easily assign an existing user to the super admin role by using the `grant_super_admin()` function. This function accepts only

one parameter, which is the user ID to which you want to grant super admin privileges.

```
<?php
$user_id = 4;
grant_super_admin( $user_id );
?>
```

As quickly as you can grant super admin privileges, you can just as easily revoke them using the `revoke_super_admin()` function. This function also accepts only one parameter, which is the user ID to revoke as super admin.

```
<?php
$user_id = 4;
revoke_super_admin( $user_id );
?>
```

Both of these functions are located in `wp-admin/includes/ms.php`. This means these functions by default are not available on the public side of your site and can be used only on the admin side. For example, if you tried calling either of these functions with a shortcode, you would get a `Call to Undefined Function PHP` error.

To list all super admins in Multisite, use the `get_super_admins()` function. This function returns an array of all super admin usernames in your network.

```
<?php
$all_admins = get_super_admins();
print_r( $all_admins );
?>
```

This will return the following array of super admins:

```
Array
(
    [0] => admin
    [1] => brad
)
```

You can also easily check specific users' IDs to determine whether they are super admins in your network. To do so, use the `is_super_admin()` function.

```
<?php
$user_id = 1;

if ( is_super_admin( $user_id ) ) {
    echo 'User is Super admin';
}
?>
```

Checking the Site Owner

Every site in your Multisite network has a site owner. This owner is defined by the admin email address stored in the site options and is set when a new site is created in your network. If you allow open site registration, the user who created the site will be set as the site owner. If you created the site in the dashboard, you can set the owner's email at time of creation.

In some cases, you may want to retrieve a site owner and corresponding user data. The following is an example of how you can do:

```
<?php
$site_id = 3;
$admin_email = get_blog_option( $blog_id, 'admin_email' );
$user_info = get_user_by( 'email', $admin_email );
print_r( $user_info );
?>
```

First, use the `get_blog_option()` function to retrieve the `admin_email` value for blog ID 3. Next use the `get_user_by()` function to retrieve the user data based on the admin email. This function enables you to retrieve user data by either user ID, slug, email, or login. In this case, use the admin email to load the user data. The results are shown here:

```
WP_User Object
(
    [ID] => 3
    [user_login] => freddy
    [user_pass] => $P$B0VRNh0UbN/4YqMFB8fl30ZM2FGKfg1
    [user_nicename] => Freddy Krueger
    [user_email] => freddy@example.com
    [user_url] =>
    [user_registered] => 2019-10-31 19:00:00
    [user_activation_key] =>
    [user_status] => 0
    [display_name] => Freddy
    [spam] => 0
)
```

```

[deleted] => 0
[first_name] => Freddy
[last_name] => Krueger
[nickname] => fredster
[description] =>
[rich_editing] => true
[comment_shortcuts] => false
[admin_color] => fresh
[use_ssl] => 0
[aim] =>
[yim] =>
[jabber] =>
[source_domain] => example.com
[primary_blog] => 3
[wp_3_capabilities] => Array
(
    [administrator] => 1
)

[wp_3_user_level] => 10
[user_firstname] => Freddy
[user_lastname] => Krueger
[user_description] =>
)
)

```

As you can see, a lot of useful user information is returned for the site admin account.

Network Stats

Multisite features a few functions to generate stats about your network. The `get_user_count()` function returns the total number of users registered in WordPress. The `get_blog_count()` function returns the total number of sites in your network. You can also use the `get_sitestats()` function to retrieve both values at once in an array.

```

<?php
$user_count = get_user_count();
echo '<p>Total users: ' . $user_count . '</p>';

$blog_count = get_blog_count();
echo '<p>Total sites: ' . $blog_count . '</p>';

$network_stats = get_sitestats();
print_r( $network_stats );
?>

```

DATABASE SCHEMA

WordPress Multisite features a different database schema from standard WordPress. When updating or enabling Multisite, WordPress creates the necessary tables in your database to support Multisite functionality.

Multisite-Specific Tables

WordPress stores global Multisite settings in centralized tables. These tables are installed only when Multisite is activated and installed, excluding `wp_users` and `wp_usermeta`.

- `wp_blogs`: Stores each site created in Multisite
- `wp_blogmeta`: Stores various metadata at the global level for any sites wanting to store data here
- `wp_registration_log`: Keeps a log of all users registered and activated in WordPress
- `wp_signups`: Stores users and sites registered using the WordPress registration process
- `wp_site`: Stores the primary site's address information
- `wp_sitemeta`: Stores various metadata for every network
- `wp_users`: Stores all users registered in WordPress
- `wp_usermeta`: Stores all metadata for user accounts in WordPress

The rest of the tables created for Multisite are site-specific tables.

Site-Specific Tables

Each site in your network features site-specific database tables. These tables hold the content and setting specific to that individual site. Remember, these tables are prefixed with the `$table_prefix` value defined in `wp-config.php`, followed by `$blog_id` and then the table name.

- `wp_1_commentmeta`
- `wp_1_comments`

- wp_1_options
- wp_1_postmeta
- wp_1_posts
- wp_1_terms
- wp_1_termmeta
- wp_1_term_relationships
- wp_1_term_taxonomy

As you can see, these tables can make your database quickly grow in size. That's why the only limitation to WordPress Multisite is the amount of server resources you have available to power your network of sites. If your network contains 1,000 sites, your database would have more than 9,000 tables. If you have plugins active that create their own database tables for each site, you would add those here as well.

Obviously, this wouldn't be ideal to host on a small shared hosting account. Shared web hosts literally share server resources for multiple websites and applications on single servers, and they are likely to have a low tolerance for customers who have thousands of database tables where there are usually only nine.

QUERY CLASSES

Multisite comes complete with query classes and matching single object classes to make querying for and interacting with these objects feel familiar. If you remember using `WP_Query`, covered in [Chapter 5](#), you will feel right at home.

`WP_Site_Query`

This is the class you will use when you need to query the `wp_blogs` database table for site data. Even the `get_blog_details()` function from earlier in this chapter uses it internally.

This class is fully documented in a modern format in the official WordPress code documentation at

[developer.wordpress.org/reference/classes/wp_site_query.](https://developer.wordpress.org/reference/classes/wp_site_query/)

WP_Network_Query

This is the class you will use when you need to query the `wp_sites` database table for network data. Functions like `get_networks()` use this query class very early due to how WordPress is loaded to ensure that sites are queried for correctly.

This class is fully documented in a modern format in the official WordPress code documentation at

[developer.wordpress.org/reference/classes/wp_network_query.](https://developer.wordpress.org/reference/classes/wp_network_query/)

OBJECT CLASSES

Similar to query classes, Multisite comes with single object classes for sites and networks to make interacting with these objects easy. If you remember using `WP_Post` in [Chapter 5](#), this will feel familiar.

WP_Site

This is the object class that is returned by `WP_Site_Query` and corresponds to a single row in the `wp_blogs` database table.

This class is fully documented in a modern format in the official WordPress code documentation at

[developer.wordpress.org/reference/classes/wp_site.](https://developer.wordpress.org/reference/classes/wp_site/)

WP_Network

This is the object class that is returned by `WP_Network_Query` and corresponds to a single row in the `wp_sites` database table.

This class is fully documented in a modern format in the official WordPress code documentation at

[developer.wordpress.org/reference/classes/wp_network.](https://developer.wordpress.org/reference/classes/wp_network/)

SUMMARY

WordPress Multisite features limitless possibilities. Enabling Multisite opens the door to creating an amazing network of sites. This also opens up new doors for your plugins with the additional Multisite features and functions.

When developing plugins for WordPress, you need to test your plugins in a Multisite setup to verify that they are compatible. Multisite is included in every WordPress download, and many users convert their standard site installation to Multisite to take advantage of the rapid site deployment features and network capabilities. It will always be important to make sure your plugins work with Multisite, even if it is not a feature you use yourself.

14

The Kitchen Sink

WHAT'S IN THIS CHAPTER?

- Understanding the WP_Query class and the loop
- Creating shortcodes for users to output dynamic content
- Building widgets for users to add to sidebars
- Registering custom dashboard widgets
- Adding custom rewrite rules
- Sending, receiving, and processing data via the Heartbeat API

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are at

www.wiley.com/go/prowordpressdev2e on the Downloads tab.

WordPress has changed a lot over the years. In the first edition of this book, many of the sections in this chapter had their own chapters. However, some of these topics are not as important to WordPress plugin development today as they once were. Other sections of this chapter cover smaller things that did not have a home anywhere else in this book.

In this chapter, you will be getting the “kitchen sink”; everything and anything not covered elsewhere is here. Much of this chapter will be high-level overviews of topics you may find yourself exploring further on your own.

QUERYING AND DISPLAYING POSTS

Generally speaking, WordPress will automatically query posts from the database on a given page view. Then, the user's theme will display the posts in the layout the theme designer has chosen. Theme development is outside

the scope of what you are learning from this book on developing plugins. However, there is some crossover, and you will need to query and display posts from time to time.

In this section of the chapter, you will learn some use cases for displaying posts in the context of the plugin, get an overview of how to query posts from the database, and learn about the all-important loop for outputting posts. At the end of the day, posts are just data. This means you could do anything you need with the data and not be limited to the examples presented here.

Use Case for Displaying Posts

As you learned in [Chapter 8](#), “Content,” you can create any post type. Therefore, your plugins are not limited to simply displaying blog posts. In that chapter, you built a “book” post type for your plugin user to store their book collection. Imagine the many ways you could potentially output such a collection.

The following are examples of how you might want to output book posts:

- Display recently added book titles in a standard list
- Create a gallery of thumbnails for books, a bookshelf of sorts
- Generate a list of books based on their genre
- Output a random selection of books from the collection to visitors

There are also various ways that you can provide the user access for outputting these things:

- Use a PHP function to manually put in a theme template (not user-friendly)
- Build a widget (see the “Widgets” section of this chapter)
- Create a shortcode (see the “Shortcodes” section of this chapter)
- Add a block for the block editor (see [Chapter 7](#), “Blocks and Gutenberg”)

Because WordPress is transitioning toward everything being block-based in the coming years, building a custom block is the ideal method to present such options to the user. However, having a backup option, such as a widget or shortcode, makes sense too. Not all users are currently using the block editor. If building a public plugin, it can be nice to offer a range of options for your plugin's users.

WP_Query Overview

To display posts, you must first query them from the database. WordPress has several functions for querying posts. However, they all eventually run through the `WP_Query` class. Most of the time, you will want to directly interact with this class instead of using one of the wrapper functions. It will keep things simple.

Take a look at the constructor method for the class:

```
<?php
WP_Query::__construct( string|array $query = '' );
```

The method takes a single parameter of `$query`, which can be a query string or array of parameters for querying posts. There are dozens of possible parameters, which can be found in the developer documentation:

https://developer.wordpress.org/reference/classes/wp_query/#parameters.

The simplest way to query a set of blog posts is the following:

```
<?php
$query = new WP_Query();
```

That code will return an object that includes the number of posts set in the user's post-per-page setting via the Reading Settings admin screen. By default, this is 10 posts.

Suppose you wanted to query five “book” posts in alphabetical order. You would need to set the `post_type`, `posts_per_page`, `order`, and `orderby` parameters.

```
<?php
$query = new WP_Query( [
    'post_type'      => 'book',
    'posts_per_page' => 5,
```

```
        'order'          => 'ASC',
        'orderby'        => 'title'
    ] );
```

While there are dozens of possible parameters, these four are some of the most commonly used.

Querying posts from the database does not actually display posts. What it does is set up the posts so that you can then use the returned object how you want.

The Loop

The *loop* in WordPress is just a fancy of way of saying to check whether any posts are available from a post query and to loop through them. Technically, the actual loop is a PHP `while` loop that loops through the found posts. However, the terminology usually colloquially refers to the `if` statement surrounding the loop too.

A typical loop looks like the following within a theme template:

```
<?php if ( have_posts() ) : ?>

    <?php while ( have_posts() ) : the_post(); ?>

        <!-- Post is output here. -->

    <?php endwhile; ?>

<?php endif; ?>
```

It is not the most elegant code for displaying posts, but this method has been a part of the platform for well over a decade. In the preceding code, there are two primary functions in use: `have_posts()` and `the_post()`. The former serves as a conditional for the `if` statement and iterator for the `while` loop. The latter sets up the current post's data via the global `$post` variable, which is an instance of a `WP_Post` object.

Developers familiar with other templating systems will likely find this method of displaying posts a little old-school. However, WordPress tends to favor backward compatibility above all else.

Both `have_posts()` and `the_post()` are wrapper functions for `WP_Query` methods and should be used only in the main query and loop for a page. In the context of a plugin, you will reference the methods as `$query->have_posts()` and `$query->the_post()`, respectively. Take a look at the following example to see what the loop might look like in a plugin:

```
<?php
if ( $query->have_posts() ) {

    while ( $query->have_posts() ) {
        $query->the_post();
    }
}

// Reset post data.
wp_reset_postdata();
```

You probably noticed that an extra function call to `wp_reset_postdata()` was appended to the previous code example. This is also another peculiarity about displaying posts in WordPress. Any time you create a custom query and loop through the posts, you need to reset the global post data.

Remember that `$query->the_post()` sets up a global `$post` variable? That must be reset to the original page's post data, and the `wp_reset_postdata()` function does that. Failing to run this reset could mean that your plugin stomps on data, which may result in breaking the current global posts query or causing other plugins to not work.

If you remember some of these basic rules and gotchas, you will be OK. It's not the prettiest system ever invented, but it does get the job done.

In the previous section on querying posts, you built a query for grabbing the five most recent books added to the site in alphabetical order. Take that query and combine it with a loop for outputting the book titles as a list.

```
<?php
// Query 5 books in alphabetical order.
$query = new WP_Query( [
    'post_type'          => 'book',
    'posts_per_page'    => 5,
    'order'              => 'ASC',
    'orderby'             => 'title'
] );
```

```

// Check if there are any posts before output.
if ( $query->have_posts() ) {

    // Open list tag if posts are found.
    echo '<ul>';

    // Loop through the found posts.
    while ( $query->have_posts() ) {
        $query->the_post();

        // Display the book title as list items.
        the_title( '<li>', '</li>' );
    }

    // Close list tag.
    echo '</ul>';
}

// Reset post data.
wp_reset_postdata();

```

The preceding code does everything you learned. You first query the posts you want. Then, you check whether any posts were found. If so, you loop through the found posts and display them. The only new concept the code presented was the function `the_title()`, which is a basic template tag that outputs the current post's title (global `$post->post_title`).

Now you will build a blog post query and loop that displays posts in a more common `<article>` wrapper element. This query will list the five most recent posts by date.

```

<?php
$query = new WP_Query( [
    'post_type'      => 'post',
    'posts_per_page' => 5,
    'order'          => 'DESC',
    'orderby'         => 'date'
] );

// Check if there are any posts before output.
if ( $query->have_posts() ) {

    // Loop through the found posts.
    while ( $query->have_posts() ) {
        $query->the_post(); ?>

```

```

<article class="post">
    <h2 class="post-title">
        <a href="<?php the_permalink(); ?>"><?php
the_title(); ?></a>
    </h2>

    <div class="post-content">
        <?php the_content(); ?>
    </div>
</article>

<?php }
}

// Reset post data.
wp_reset_postdata();

```

As you can see, nothing is different about the basic query and loop code other than the parameters passed to `WP_Query`. The real difference is with the HTML output, which uses template tags for outputting post data. Template tags are basic PHP functions generally reserved for themes, but you can use them when needed in plugins. The full list of template tags is available at

<https://developer.wordpress.org/themes/references/list-of-template-tags>.

In the next section, you will learn how to turn custom queries and post loops into a practical example via the Shortcode API.

SHORTCODES

Shortcodes are WordPress-specific code that enables you to do nifty things with little effort, such as embed content or create objects that would normally require code that is too complicated for the average end user to piece together. In this section, you'll learn how to create simple shortcodes to allow your plugin users to enhance their posts.

WARNING *Shortcodes are on their way out. While they are simple to build and will likely be in WordPress forever for backward compatibility, they are essentially outdated now. You should consider building a block instead, as shown in [Chapter 7](#). Shortcodes should be available only as a fallback for users who are still using the classic editor.*

What Shortcodes Are

The Shortcode API enables creating simple macro codes, sometimes also referred to as *bbcodes* for their similarity with a popular syntax in various forums and bulletin boards.

Essentially, a shortcode is a simple tag syntax between square brackets, such as `[example]`, used in post content. When rendered on the front end, the shortcode is dynamically replaced with a more complex and user-defined content.

Out of the box, WordPress registers several shortcodes that you can use. For example, when you upload multiple images to a post, you can drop the `[gallery]` shortcode in the post editor. This shortcode will be replaced with a nicely formatted gallery of your images when viewing the post on the front end.

WordPress out of the box registers shortcodes you can use. When you upload multiple images attached to a given post, you can simply insert `[gallery]` in your post, and this shortcode will be replaced with a nicely formatted gallery of your images.

The following are the default shortcodes registered by WordPress. You should consider these shortcode tag names as off-limits and not register a tag with the same name.

- `wp_caption`
- `caption`
- `gallery`
- `playlist`

- audio
- video
- embed

When registering a shortcode, you should follow a few rules. First, it is good practice to always prefix the shortcode tag with a prefix unique to your plugin. For the purposes of the shortcodes in this chapter, you will use `pdev_`. Therefore, a custom gallery shortcode might be `[pdev_gallery]`.

You must also avoid square braces, angle braces, ampersands, forward slashes, whitespace, nonprintable characters, and quotation marks. Even hyphens can cause issues if the pre-hyphen characters match between two different shortcodes, which causes a conflict between the two. In general, it is best to stick with alphanumeric characters and underscores in shortcode tags.

Here are some examples of shortcodes that follow best practices:

- `[pdev_post_list]`
- `[pdev_featured_image]`
- `[pdev_lightbox_gallery]`

Basically, don't go out of your way to make difficult names. The idea is that shortcodes should be “short” and easy-to-remember tags that users can drop into their post content. Don't make it hard for them or yourself.

Register Custom Shortcodes

In this section, you will learn how to register shortcodes. WordPress provides a simple function named `add_shortcode()` that allows plugin authors to register custom shortcodes.

```
<?php
add_shortcode( string $tag, callable $callback );
```

The function accepts two parameters.

- `$tag`: The shortcode tag name covered in the previous section

- `$callback`: A callback function or class method that renders the shortcode output on the front end

In most cases, shortcodes should be registered on WordPress' `init` action hook. The following is a quick example of registering an example shortcode that outputs a simple “Hello, world.” message:

```
<?php
add_action( 'init', function() {

    add_shortcode( 'pdev_hello_world', function() {
        return 'Hello, world.';
    } );
} );
```

NOTE *Shortcode callback functions should always return a string. They should never echo, print, or otherwise directly output content. This is because shortcodes are parsed before their content is output. Therefore, always return a string and let WordPress output the content at the appropriate time.*

Building a Simple Shortcode

Now, start thinking about a more practical example. Remember how you output a post list in the “Querying and Displaying Posts” section of this chapter? That would make for a perfect example of a simple shortcode.

Start by defining what posts you want to query and display. Keeping this simple, grab the last 10 posts published to the site. Then, you create an unordered HTML list of the posts.

Now create a new plugin file named `plugin-post-list-shortcode.php` with the following code:

```
<?php
/**
 * Plugin Name: Post List Shortcode
 * Plugin URI: http://example.com/
 * Description: Output the last 10 posts via the
 [pdev_post_list] shortcode.
 * Author: WROX
```

```
* Author URI:  http://wrox.com
*/
add_action( 'init', 'pdev_register_post_list_shortcodes' );

// Register shortcodes.
function pdev_register_post_list_shortcodes() {
    add_shortcode( 'pdev_post_list',
'pdev_post_list_shortcode' );
}

// Shortcode callback function.
function pdev_post_list_shortcode() {

    // Create empty string for content.
    $html = '';

    // Query last 10 posts.
    $query = new WP_Query( [
        'post_type'      => 'post',
        'posts_per_page' => 10,
        'order'          => 'DESC',
        'orderby'         => 'date'
    ] );

    // Check if there are any posts before output.
    if ( $query->have_posts() ) {

        // Open list tag if posts are found.
        $html .= '<ul>';

        // Loop through the found posts.
        while ( $query->have_posts() ) {
            $query->the_post();

            // Add list item with linked post title.
            $html .= sprintf(
                '<li><a href="%s">%s</a></li>',
                esc_url( get_permalink() ),
                the_title( '', '', false )
            );
        }

        // Close list tag.
        $html .= '</ul>';
    }

    // Reset post data.
}
```

```
wp_reset_postdata();

// Return shortcode HTML.
return $html;
}
```

The big difference with outputting a post list and creating a shortcode for handling a post list is that the shortcode callback must return a string instead of output. As you can see in the preceding plugin code, everything to be output was assigned to the `$html` variable and returned in the shortcode callback.

After activating the plugin, add the following code to any post via the post-editing screen in the admin:

```
[pdev_post_list /]
```

That will output a simple post list, as shown in [Figure 14-1](#).

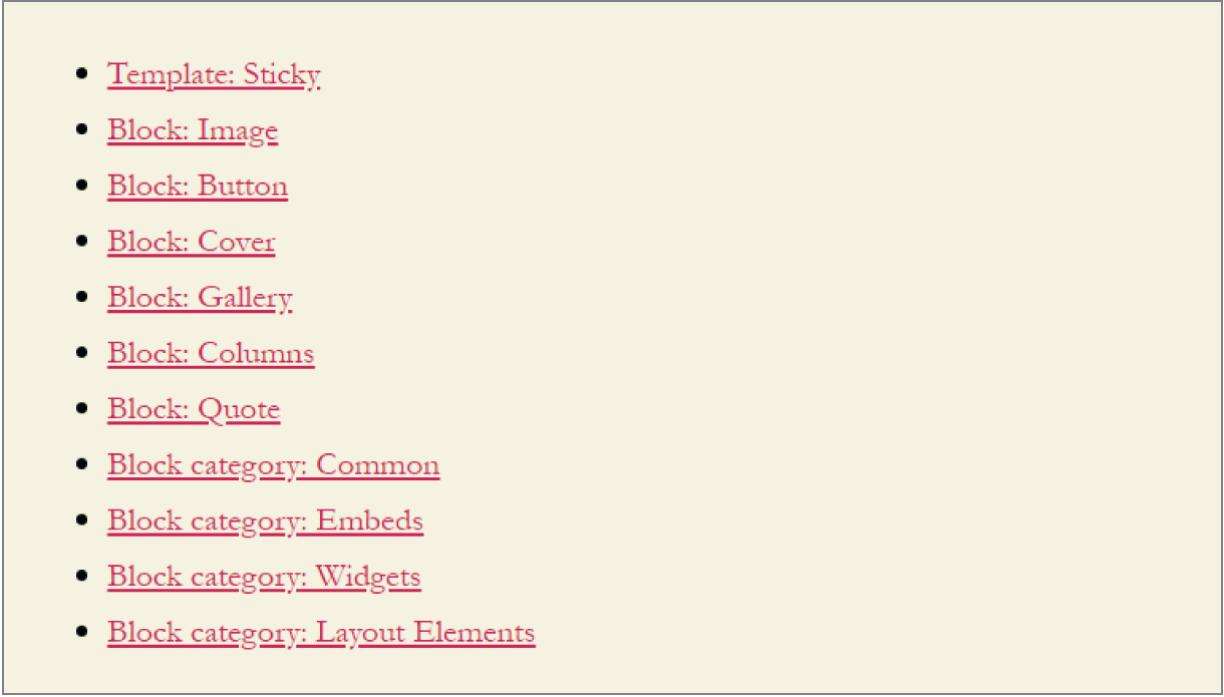
- 
- The image shows a screenshot of a WordPress post editor. The content area is a light yellow color. On the left, there is a list of block types, each preceded by a red bullet point. The blocks listed are: Template: Sticky, Block: Image, Block: Button, Block: Cover, Block: Gallery, Block: Columns, Block: Quote, Block category: Common, Block category: Embeds, Block category: Widgets, and Block category: Layout Elements.
- [Template: Sticky](#)
 - [Block: Image](#)
 - [Block: Button](#)
 - [Block: Cover](#)
 - [Block: Gallery](#)
 - [Block: Columns](#)
 - [Block: Quote](#)
 - [Block category: Common](#)
 - [Block category: Embeds](#)
 - [Block category: Widgets](#)
 - [Block category: Layout Elements](#)

FIGURE 14-1: Simple post list

Building a Shortcode with Parameters

The shortcode in the previous section did not provide the user with much control. It can only ever output a simple post list. However, the shortcode

system allows you to create custom parameters so that users can modify what the shortcode outputs.

When accepting parameters, they will be passed as an array of arguments as the first parameter of the shortcode callback, which would look like the following:

```
<?php
function pdev_shortcode_callback( $attr ) {
    return '';
}
```

When accepting attributes, you should always parse them against an array of defaults because users will not always pass along data for each.

WordPress provides the `shortcode_atts()` function for parsing the attributes.

```
<?php
shortcode_atts( array $pairs, array $attr, string $shortcode
= '' );
```

The function returns the parsed attributes. It accepts three parameters.

- `$pairs`: The list of supported attributes and their default values.
- `$attr`: The attributes passed into the shortcode callback to merge with the defaults.
- `$shortcode`: The name of the shortcode. If provided, it creates a filter hook named `shortcode_atts_{$shortcode}` that other plugin developers may filter.

WARNING *All user-defined parameters must be sanitized. Like any other unknown data, you must consider that it is malicious. See [Chapter 4](#), “Security and Performance,” for more information on securing input data.*

Now you will build a `[pdev_custom_posts]` shortcode that outputs a list of posts just like the `[pdev_post_list]` shortcode from the previous section. However, the new shortcode will allow the user to define the `post_type`, `posts_per_page`, `order`, and `orderby` parameters.

Create a new plugin-custom-posts-shortcode.php file with the following code:

```
<?php
/**
 * Plugin Name: Custom Posts Shortcode
 * Plugin URI:  http://example.com/
 * Description: Output posts via the [pdev_custom_posts]
shortcode.
 * Author:      WROX
 * Author URI:  http://wrox.com
 */

add_action( 'init', 'pdev_register_custom_posts_shortcodes'
);

// Register shortcodes.
function pdev_register_custom_posts_shortcodes() {
    add_shortcode( 'pdev_custom_posts',
'pdev_custom_posts_shortcode' );
}

// Shortcode callback function.
function pdev_custom_posts_shortcode( $attr ) {

    // Create empty string for content.
    $html = '';

    // Parse attributes.
    $attr = shortcode_atts(
        [
            'post_type'      => 'post',
            'posts_per_page' => 10,
            'order'          => 'DESC',
            'orderby'         => 'date'
        ],
        $attr,
        'pdev_custom_posts'
    );

    // Sanitize attributes before proceeding.

    // Make sure post type exists.
    $attr['post_type'] = post_type_exists(
        $attr['post_type'] )
            ? $attr['post_type']
            : 'post';
```

```

// Posts per page should be an integer.
$attr['posts_per_page'] = intval(
$attr['posts_per_page'] );

// Only allow ascending or descending.
$attr['order'] = in_array( $attr['order'], [ 'ASC',
'DESC' ] )
    ? $attr['order']
    : 'DESC';

// Strip tags from orderby.
$attr['orderby'] = wp_strip_all_tags( $attr['orderby'] )
);

// Query last 10 posts.
$query = new WP_Query( [
    'post_type'      => $attr['post_type'],
    'posts_per_page' => $attr['posts_per_page'],
    'order'          => $attr['order'],
    'orderby'         => $attr['orderby']
] );

// Check if there are any posts before output.
if ( $query->have_posts() ) {

    // Open list tag if posts are found.
    $html .= '<ul>';

    // Loop through the found posts.
    while ( $query->have_posts() ) {
        $query->the_post();

        // Add list item with linked post title.
        $html .= sprintf(
            '<li><a href="%s">%s</a></li>',
            esc_url( get_permalink() ),
            the_title( '', '', false )
        );
    }

    // Close list tag.
    $html .= '</ul>';
}

// Reset post data.
wp_reset_postdata();

```

```
    // Return shortcode HTML.
    return $html;
}
```

As you can see, the code is similar to the earlier post list shortcode you built. The major difference is that it accepts several parameters and parses them with a set of defaults via the `shortcode_atts()` function. Then, each parameter is sanitized to make sure it is safe and/or valid.

When you enter the following into the post editor, it will list five pages in alphabetical order:

```
[pdev_custom_posts post_type="page" posts_per_page="5"
order="ASC"
orderby="title" ]
```

End users can mix and match which parameters they use or even choose not to use any of the parameters. You could take this example of a simple post list shortcode even further and allow users to set more parameters if you want. However, keep in mind that shortcodes should be simple to use.

Building a Shortcode with Content

In the two previous shortcode examples, you built what are generally called *self-closing shortcodes*, meaning that the shortcode closes itself with a / character before the closing] bracket. Some shortcodes can accept content within. Take a look at the following examples to see the difference:

```
// Self-closing shortcode:
[pdev_example_1 /]

// Shortcode with content:
[pdev_example_2]Example content.[/pdev_example_2]
```

Not all shortcodes must accept content, but it is an option if you need the functionality for your plugin. If allowing content for your plugin's shortcode, the content is passed as the second parameter for the shortcode callback function.

Suppose you wanted to create a `pullquote` function for allowing users to define a quote in their post. You decide to build a `[pdev_pullquote]` shortcode to handle formatting the HTML for your plugin users.

Create a new file named `plugin-pullquote-shortcode.php` with the following code:

```
<?php
/**
 * Plugin Name: Pullquote Shortcode
 * Plugin URI:  http://example.com/
 * Description: Outputs a quote via the [pdev_pullquote]
shortcode.
 * Author:      WROX
 * Author URI:  http://wrox.com
 */

add_action( 'init', 'pdev_register_pullquote_shortcodes' );

// Register shortcodes.
function pdev_register_pullquote_shortcodes() {
    add_shortcode( 'pdev_pullquote',
'pdev_pullquote_shortcode' );
}

// Shortcode callback function.
function pdev_pullquote_shortcode( $attr, $content = '' ) {

    // Bail if there is no content.
    if ( ! $content ) {
        return '';
    }

    // Return formatted content.
    return sprintf(
        '<blockquote class="pdev-
pullquote">%s</blockquote>',
        wpautop( wp_kses_post( $content ) )
    );
}
```

The shortcode callback function from the preceding code does two things. First, it checks whether there is any content because this particular shortcode requires the user to enter some content. If no content is passed in, it returns an empty string.

If the user does define custom content, that content is sanitized and wrapped in custom HTML. You could take this concept further and add custom styles or allow for custom attributes as you learned in the previous section.

Now try adding the following to the post editor:

```
[pdev_pullquote]
"Professional WordPress Plugin Development" is the greatest
book ever!
[/pdev_pullquote]
```

On the front end of the site, WordPress will properly output the shortcode as the following HTML:

```
<blockquote class="pdev-pullquote">
  <p>"Professional WordPress Plugin Development" is the
  greatest book ever!</p>
</blockquote>
```

Shortcode Tips

Shortcodes are a great way to put a lot of power in the hands of average users and allow them to output complex and dynamic content. To guarantee the best end-user experience, you must keep in mind two important principles as a plugin author.

- Make things simple and foolproof.
- Remember that shortcodes are dynamic.

Think Simplicity for the User

As a plugin user, it is awesome to add new features to one's blog and to be able to write a simple shortcode that outputs more complex content. However, it can be cumbersome to remember the shortcode parameter syntax. Often, users end up with the impression that they are learning a new markup language.

Remember the `[pdev_custom_posts]` shortcode earlier in this chapter? It allows users to input between one and four parameters. You will have to decide whether all of the options are too much for your users.

This may also be different depending on what your goal is. Do you really want to simply offer a post or page list option? You may consider splitting the shortcode into separate `[pdev_post_list]` and `[pdev_page_list]` shortcodes. You may also cut back on parameters. For example, are the `order` and `orderby` parameters necessary?

Allowing lots of options is neat, but you don't want your users constantly checking the plugin documentation. This can make for a bad experience. Instead, make things simple and foolproof so that users can use your plugin instinctively.

Remember That Shortcodes Are Dynamic

_shortcode output is dynamically generated. This means that every time WordPress displays a post, the post content is parsed, and the shortcodes are replaced with the returned result of their callback function.

Replacements like those you coded in this chapter are lightning fast. You don't have to worry about WordPress' performance when you register new shortcodes.

Performance should be a consideration if your shortcodes pull information either from the database or from remote websites.

- In the first case, your code will issue extra SQL queries, which can hinder performance on slow web hosts.
- In the second case, your shortcode will perform external HTTP requests that could slow down the whole page rendering, while WordPress is awaiting the remote server response to parse.

In such cases, you should consider caching the result of your shortcode. See [Chapter 4](#) for more information on caching data.

Look under the Hood

Besides using `add_shortcode()` to register new shortcodes, the Shortcode API has other interesting functions and capabilities. In this section, you will learn how to make use of these extra features.

`remove_shortcode()`

You can dynamically unregister a shortcode using the `remove_shortcode()` function. It is important to note that this function must be called after the shortcode has been registered for it to work.

```
remove_shortcode( 'shortcode_tag_name' );
```

remove_all_shortcodes()

Similarly, you can dynamically unregister all registered shortcodes using the `remove_all_shortcodes()` function. Technically, this function simply resets the global `$shortcode_tags` to an empty array.

strip_shortcodes()

The function `strip_shortcodes()` strips registered shortcodes from string content. Imagine the following content with the `[pdev_example]` shortcode. It will be stripped from output.

```
<?php
$content = 'Hello, world! [pdev_example]';

echo strip_shortcodes( $content );
```

do_shortcode()

The `do_shortcode()` function searches string content passed into it for shortcodes and processes them. By default, WordPress uses `do_shortcode()` as a filter on the post content (`the_content` filter hook), which is why shortcodes in post content are automatically processed.

However, you are not limited to just the post content. Plugins and even themes can create “shortcode-aware” areas via custom settings if they want.

Imagine your plugin had a custom database option named `pdev_custom_content` (see [Chapter 3](#), “Dashboard and Settings,” for information on how to create plugin settings). You want to allow the user to enter some basic text and also allow for shortcodes. On output of the option, you merely need to wrap the database value in the `do_shortcode()` function.

```
<?php
$content = get_option( 'pdev_custom_content', 'Some default
text.' );

echo do_shortcode( $content );
```

You can take this even further if you want and have shortcodes that are able to process other shortcodes. Earlier in this chapter, you learned how to build

a shortcode with content. It is possible for that shortcode's content to also have shortcodes.

Consider the following post content with nested shortcodes:

```
[pdev_example]This shortcode has [another_shortcode /]  
within it.[/pdev_example]
```

For the [another_shortcode] tag to be processed, the [pdev_example] shortcode must allow for it by running its content through `do_shortcode()`. The following code snippet shows how this is possible:

```
<?php  
add_shortcode( 'pdev_example', function( $attr, $content =  
' ' ) {  
    return do_shortcode( $content );  
} );
```

By default, WordPress will not process nested shortcodes. The preceding code is the only way to allow for it. At some point in the shortcode callback function, the returned content must be run through `do_shortcode()`.

WIDGETS

Widgets are a great way to give the users of your plugin an easy method to display your plugin information or data. WordPress features a Widgets API for creating and interacting with widgets. In this section, you explore how to create widgets, add and save widget options, and display plugin information in the widget output.

NOTE *The future for widgets is unknown. Like shortcodes, it's possible that they will eventually be replaced entirely by the block system described in [Chapter 7](#). At the time this book is published, WordPress is in a transitional phase where the block system is moving toward handling full-site editing. For now, widgets are still a useful feature of the platform.*

Creating a Widget

You create all widgets in WordPress by extending the `WP_Widget` class. To understand how the widget class works, it's helpful to look at an overview of what the typical widget class looks like.

```
<?php
namespace PDEV;

use WP_Widget;

class MyWidget extends WP_Widget {

    // Sets up the widget.
    public function __construct() {}

    // Displays the widget.
    public function widget() {}

    // Outputs widget options form.
    public function form() {}

    // Callback on widgets update to save options.
    public function update() {}

}
```

As you can see, the custom widget subclass features multiple methods for your widget, each with a specific purpose.

Now it's time to create a widget for your plugin. For this first widget, you create a simple text-based widget to save and display your favorite movie and song. It is a simple example that demonstrates how to save text data in a WordPress widget.

First, create a new folder for your plugin named `plugin-favorites-widget` and put a new file within it named `plugin.php`. Then, set up your plugin header.

```
<?php
/**
 * Plugin Name: Favorites Widget
 * Plugin URI: http://example.com/
 * Description: Allows the user to enter their favorite
 * movie and song.
 * Author: WROX
 * Author URI: http://wrox.com
 */
```

To register custom widgets, you use the `widgets_init` action hook. This hook is triggered after the default widgets have been registered in WordPress.

```
namespace PDEV\Favorites;

add_action( 'widgets_init', function() {

    require_once plugin_dir_path( __FILE__ ) .
'Widget.php';

    register_widget( __NAMESPACE__ . '\Widget' );
} );
```

This code attaches a simple anonymous function to the `widgets_init` hook. When the hook is fired by WordPress, your code loads a `Widget.php` class file, which will hold our custom widget. It then registers the widget with WordPress via the `register_widget()` function, which accepts a single parameter of the widget's class name.

NOTE *When registering a widget class that is namespaced, as shown in the preceding code, you must use the fully qualified class name when calling `register_widget()`. WordPress will not automatically know the namespace.*

Now that you have the plugin set up and are attempting to load a `Widget.php` file, go ahead and create that file within your plugin folder. You will now create the actual code to make your widget work.

You need to extend the `WP_Widget` class by creating a new class with the unique name you defined when you registered your widget. Start by using the basic template from earlier.

```
<?php
namespace PDEV\Favorites;

use WP_Widget;

class Widget extends WP_Widget {

    // Sets up the widget.
    public function __construct() {}
```

```

    // Displays the widget.
    public function widget() {}

    // Outputs widget options form.
    public function form() {}

    // Callback on widgets update to save options.
    public function update() {}
}


```

From this point, you need to set up the widget via the constructor method. Here, you will define some widget options with WordPress.

```

// Sets up the widget.
public function __construct() {

    // Set up the widget options.
    $widget_options = array(
        'classname'                  => 'pdev-favorites',
        'description'                => 'Displays your
favorite movie and song.',
        'customize_selective_refresh' => true
    );

    // Create the widget.
    parent::__construct(
        'pdev-favorites',
        'Favorites: Movie and Song',
        $widget_options
    );
}

```

There are two primary things you need to do when setting up your widget. The first is to set the `$widget_options` array. For this, you can pass a custom `classname`, add a textual description of the widget, and set the `customize_selective_refresh` option, which tells WordPress whether to provide a live preview of changes to the widget without refreshing the entire page in the customizer. Unless you are doing some advanced JavaScript stuff that interferes with selective refresh in the customizer, you should set this to `true`.

For the final step of setting up the widget, you must call the parent class' `__construct()` method. The first parameter should be a unique ID for your

widget. The second parameter is the widget title. And, the third parameter is the `$widget_options` array that you set up earlier.

Now, you can get down to the business of handling the front-end widget output. Besides the constructor method, the `widget()` method is the only method required for creating a widget. The `form()` and `update()` methods that you will add later are required only for widgets that have custom options.

Update your widget class' `widget()` method with the following code:

```
// Displays the widget.
public function widget( $sidebar, $instance ) {

    // Open the sidebar widget wrapper.
    echo $sidebar['before_widget'];

    // Output the widget title if set.
    if ( ! empty( $instance['title'] ) ) {

        // Open the sidebar widget title wrapper.
        echo $sidebar['before_title'];

        // Apply filters and output widget title.
        echo apply_filters( 'widget_title',
$instance['title'], $instance,
$this->id_base );

        // Close the sidebar widget title wrapper.
        echo $sidebar['after_title'];
    }

    // Output favorite movie and song in a list.
    echo '<ul>';

    printf(
        '<li>Favorite Movie: %s</li>',
        ! empty( $instance['movie'] ) ? esc_html(
$instance['movie'] ) : 'None'
    );

    printf(
        '<li>Favorite Song: %s</li>',
        ! empty( $instance['song'] ) ? esc_html(
$instance['song'] ) : 'None'
    );
}
```

```
echo '</ul>';

// Close the sidebar widget wrapper.
echo $sidebar['after_widget'];
}
```

The first thing you should notice is the two parameters passed into the method.

- `$sidebar`: This parameter includes necessary data about the sidebar, which has generally been registered by the theme. You will need it to output the appropriate HTML tags so that it doesn't break the theme on the front end.
- `$instance`: This parameter houses the data about the specific instance of the widget that is being output (widgets can be used multiple times, which creates separate instances). Primarily, you need this data for any widget options set by the user.

The first and last things all widgets should do is print the `$sidebar['before_widget']` and `$sidebar['after_widget']` variables, respectively. These variables are the opening and closing HTML tags for the sidebar the widget is being displayed in.

The next important thing is the widget title. Not all widgets have titles, but it is generally good practice to allow the user to decide whether they want to add a title and provide an option for this. When displaying the title, it is important to apply the core WordPress `widget_title` filter hook so that other plugins can manipulate this. Take special note of the format of this hook and make sure to pass all data as shown in the following snippet so that it is available to filters:

```
echo apply_filters( 'widget_title', $instance['title'],
$instance, $this->id_base );
```

Use the `$sidebar['before_title']` and `$sidebar['after_title']` variables before and after outputting the title, respectively. Like the widget wrapper, this will wrap the title in the registered sidebar's widget title HTML.

After outputting the title, you simply need to add in your widget's custom output. In the preceding code, you made a simple list to output the user's favorite movie and song. Remember that the `$instance` array contains all of the widget options data. So, `$instance['movie']` and `$instance['song']` are both available for use.

Now you will add a settings form for your widget. It will include fields for a title, favorite movie, and favorite song. Replace the `form()` method in your widget class with the following code:

```
// Outputs widget options form.
public function form( $instance ) {

    $instance = wp_parse_args(
        (array) $instance,
        [
            'title' => 'Favorites',
            'movie' => '',
            'song' => ''
        ]
    ); ?>

    <p>
        <label>
            Title:
            <input
                type="text"
                class="widefat"
                name="<?php echo $this->get_field_name( 'title' ); ?>" value="<?php echo esc_attr( $instance['title'] ); ?>" />
        </label>
    </p>

    <p>
        <label>
            Movie:
            <input
                type="text"
                class="widefat"
                name="<?php echo $this->get_field_name( 'movie' ); ?>" value="<?php echo esc_attr( $instance['movie'] ); ?>" />
        </label>
    </p>
}
```

```

        />
    </label>
</p>

<p>
    <label>
        Song:
        <input
            type="text"
            class="widefat"
            name="<?php echo $this-
>get_field_name( 'song' ); ?>" value="<?php echo esc_attr(
$instance['song'] ); ?>" />
    </label>
</p>
<?php }

```

The `form()` method accepts a single parameter of `$instance`. Just like the `widget()` method from earlier, this is an array of settings for this specific instance of the widget.

The first thing you should do with the `$instance` variable is to parse it against an array of defaults. This will make sure that each key/value pair is set for the `$instance` array and not result in an undefined index error in the case that a key doesn't exist.

From that point, it is simply a matter of outputting a basic HTML form. There are two major things you must do to ensure that your form is correct. The first is to add an appropriate `name` attribute to form fields for this particular instance of the widget. WordPress provides the `WP_Widget::get_field_name()` method (use as `$this->get_field_name()`) for returning the correct value for this attribute. You merely need to reference it and pass in the widget setting name as the first parameter.

The second requirement is that the `value` attribute for form fields must use the `$instance` variable. For example, if you wanted to output the correct `title` setting value, you would need to reference `$instance['title']`.

The final step of creating your widget is handling the `update()` method, which is executed when widget settings are saved. This method is important

because it is the point where you sanitize settings before they are stored in the database.

Replace the `update()` method in your widget class with the following code:

```
// Callback on widgets update to save options.
public function update( $new_instance, $old_instance ) {

    // Create empty array to store sanitized data.
    $instance = [];

    // Sanitize data from widget form.
    $instance['title'] = sanitize_text_field(
        $new_instance['title'] );
    $instance['movie'] = sanitize_text_field(
        $new_instance['movie'] );
    $instance['song'] = sanitize_text_field(
        $new_instance['song'] );

    // Return sanitized data.
    return $instance;
}
```

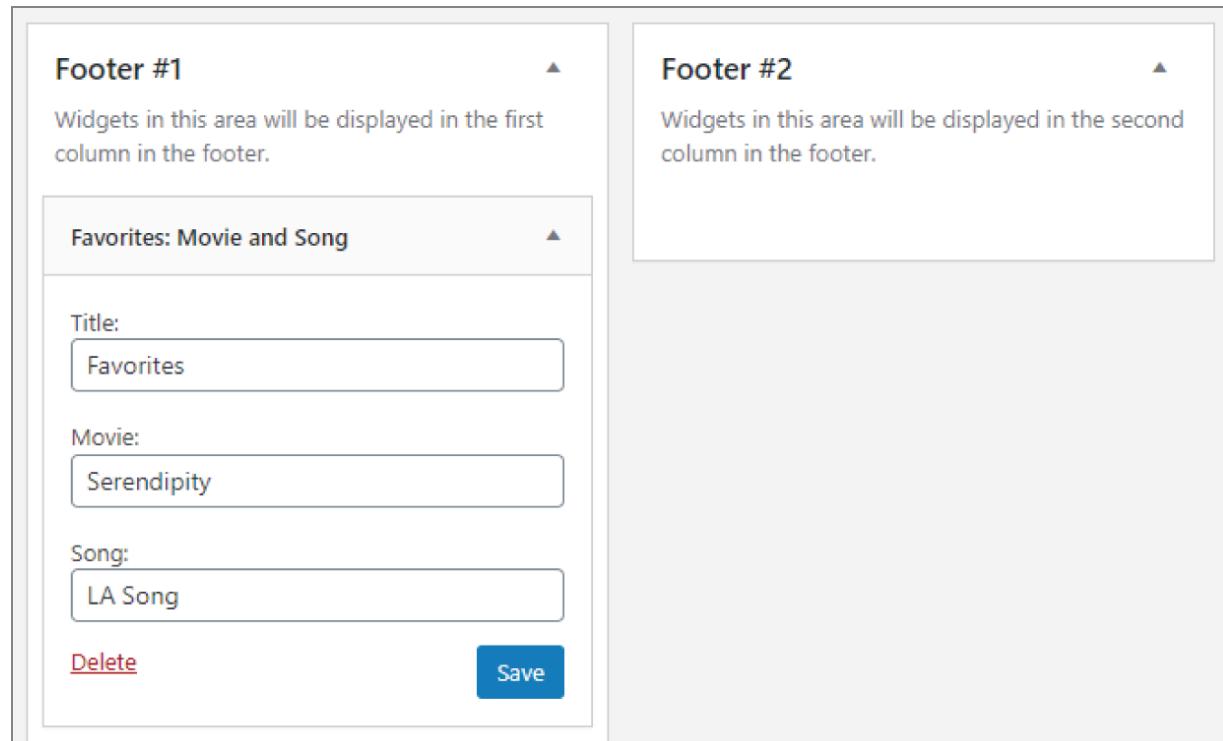
The `update()` method accepts two parameters.

- `$new_instance`: The new widget settings array set by the user before being saved to the database.
- `$old_instance`: The previously saved widgets settings. You will not need this array most of the time, but it is available if you need to work with the old data.

The first thing you should do is create a clean, empty `$instance` array that will house all of your sanitized options. You will return this array after sanitizing.

In the preceding code, the widget title, favorite movie, and favorite song settings are all sanitized via the `sanitize_text_field()` function. Because all of these options are basic text fields, this function keeps things simple. Keep in mind that you should always sanitize or validate based on the specific type of data that you are expecting. For more information on properly sanitizing data, see [Chapter 4](#).

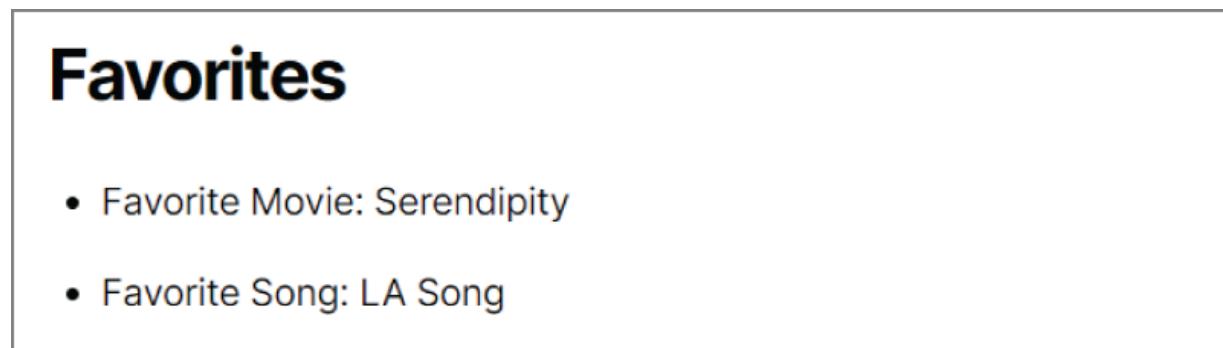
Congratulations! You have now successfully built your first widget. Take a look at what it should look like from the Widgets admin screen in [Figure 14-2](#).



The image shows the 'Widgets' admin screen. It has two columns: 'Footer #1' and 'Footer #2'. In 'Footer #1', there is a 'Favorites: Movie and Song' widget. This widget has fields for 'Title' (containing 'Favorites'), 'Movie' (containing 'Serendipity'), and 'Song' (containing 'LA Song'). At the bottom of the widget, there are 'Delete' and 'Save' buttons. In 'Footer #2', there is a blank area for widgets.

FIGURE 14-2: Widgets admin screen

After placing it in a sidebar, it should look similar to [Figure 14-3](#) on the front end.



The image shows a 'Favorites' list on a website. The title 'Favorites' is displayed in large, bold, black font. Below the title is a bulleted list: '• Favorite Movie: Serendipity' and '• Favorite Song: LA Song'.

FIGURE 14-3: Favorites list

Now review the full widget plugin code that you have put together. Your `plugin-favorites-widget/plugin.php` file should look like the following:

```

<?php
/**
 * Plugin Name: Favorites Widget
 * Plugin URI: http://example.com/
 * Description: Allows the user to enter their favorite
movie and song.
 * Author: WROX
 * Author URI: http://wrox.com
 */

namespace PDEV\Favorites;

add_action( 'widgets_init', function() {

    require_once plugin_dir_path( __FILE__ ) .
'Widget.php';

    register_widget( __NAMESPACE__ . '\\Widget' );
} );

```

The plugin-favorites-widget/Widget.php file should look like this final snippet:

```

<?php
namespace PDEV\Favorites;

use WP_Widget;

class Widget extends WP_Widget {

    // Sets up the widget.
    public function __construct() {

        // Set up the widget options.
        $widget_options = array(
            'classname' => 'pdev-favorites',
            'description' => 'Displays your favorite movie
and song.',
            'customize_selective_refresh' => true
        );

        // Create the widget.
        parent::__construct(
            'pdev-favorites',
            'Favorites: Movie and Song',
            $widget_options
        );
    }
}

```

```
}

// Displays the widget.
public function widget( $sidebar, $instance ) {

    // Open the sidebar widget wrapper.
    echo $sidebar['before_widget'];

    // Output the widget title if set.
    if ( ! empty( $instance['title'] ) ) {

        // Open the sidebar widget title wrapper.
        echo $sidebar['before_title'];

        // Apply filters and output widget title.
        echo apply_filters(
            'widget_title',
            $instance['title'],
            $instance,
            $this->id_base
        );

        // Close the sidebar widget title wrapper.
        echo $sidebar['after_title'];
    }

    // Output favorite movie and song in a list.
    echo '<ul>';

    printf(
        '<li>Favorite Movie: %s</li>',
        ! empty( $instance['movie'] ) ? esc_html(
            $instance['movie']
        ) : 'None'
    );

    printf(
        '<li>Favorite Song: %s</li>',
        ! empty( $instance['song'] ) ? esc_html(
            $instance['song']
        ) : 'None'
    );

    echo '</ul>';

    // Close the sidebar widget wrapper.
    echo $sidebar['after_widget'];
}

// Outputs widget options form.

```

```
public function form( $instance ) {

    $instance = wp_parse_args(
        (array) $instance,
        [
            'title' => 'Favorites',
            'movie' => '',
            'song' => ''
        ]
    ); ?>

    <p>
        <label>
            Title:
            <input
                type="text"
                class="widelafat"
                name="<?php echo $this->get_field_name(
'title' ); ?>" value="<?php echo esc_attr(
$instance['title'] ); ?>" />
        </label>
    </p>

    <p>
        <label>
            Movie:
            <input
                type="text"
                class="widelafat"
                name="<?php echo $this->get_field_name(
'movie' ); ?>" value="<?php echo esc_attr(
$instance['movie'] ); ?>" />
        </label>
    </p>

    <p>
        <label>
            Song:
            <input
                type="text"
                class="widelafat"
                name="<?php echo $this->get_field_name(
'song' ); ?>" value="<?php echo esc_attr(

```

```

$instance['song'] ); ?>
    />
    </label>
</p>
<?php }

// Callback on widgets update to save options.
public function update( $new_instance, $old_instance ) {

    // Create empty array to store sanitized data.
    $instance = [];

    // Sanitize data from widget form.
    $instance['title'] = sanitize_text_field(
$new_instance['title'] );
    $instance['movie'] = sanitize_text_field(
$new_instance['movie'] );
    $instance['song'] = sanitize_text_field(
$new_instance['song'] );

    // Return sanitized data.
    return $instance;
}
}

```

DASHBOARD WIDGETS

WordPress also features a Dashboard Widgets API. You can use this API to create custom widgets on the Dashboard screen in the WordPress admin. Dashboard widgets can be anything, but their purpose is generally informational to help the user in some capacity. This can be data about your plugin or perhaps an RSS feed that points to tutorials on your website.

Some plugin authors use dashboard widgets to advertise products and services. For many users, this is seen as spammy. Going overboard here can lead to a poor reputation in the community. It is best to leave any ads to your plugin's own admin screens and not place them on a screen unrelated to your plugin.

Dashboard widgets should not be confused with the Widgets API discussed earlier in this chapter. While the two share a similar name, they are nothing alike.

Creating Dashboard Widgets

To create your dashboard widget, you use the `wp_add_dashboard_widget()` function.

```
<?php
wp_add_dashboard_widget(
    string $widget_id,
    string $widget_name,
    callable $callback,
    callable $control_callback = null,
    array $callback_args = null
);
```

The function accepts the following parameters:

- `$widget_id`: The CSS ID added to the widget's HTML wrapper.
- `$widget_name`: The title displayed for the widget header.
- `$callback`: Function, method, or other callable for displaying your widget's content.
- `$control_callback`: Function, method, or other callable for outputting the form controls for the widget. This is necessary only for widgets with settings.
- `$callback_args`: Optional array of arguments to pass to the widget callback function.

To create a dashboard widget, use the `wp_dashboard_setup` action hook. This hook is executed directly after the default dashboard widgets have been initialized, but prior to them being displayed.

Now register a new dashboard widget as shown in the following snippet:

```
add_action( 'wp_dashboard_setup',
    'pdev_simple_dashboard_register' );

// Registers dashboard widgets.
function pdev_simple_dashboard_register() {

    wp_add_dashboard_widget(
        'pdev-simple-dashboard',
        'Plugin: Report Bugs',
```

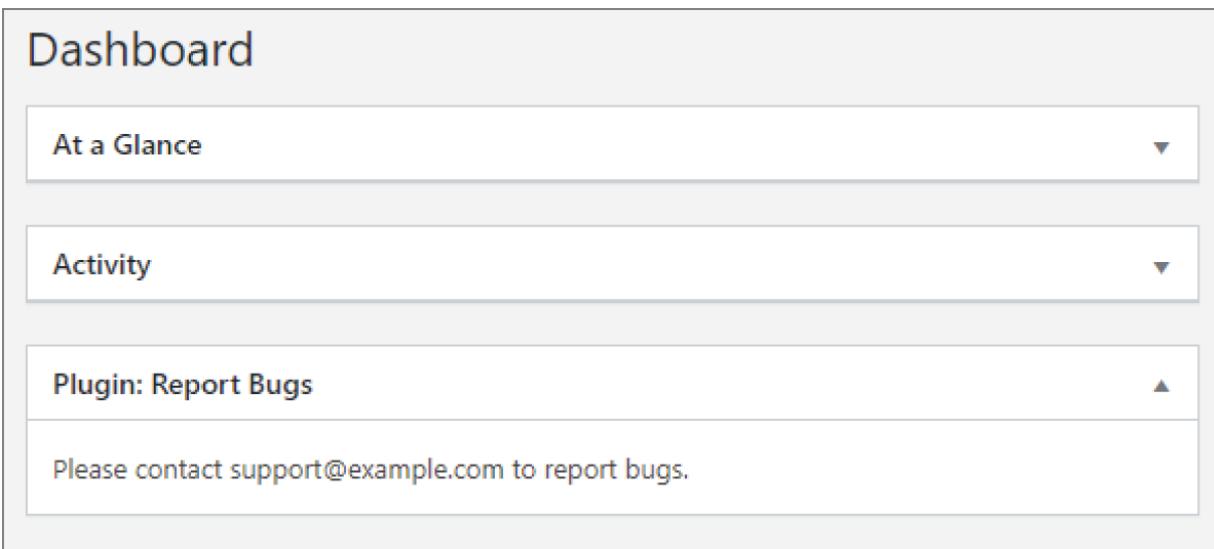
```
        'pdev_simple_dashboard_display'
    );
}
```

The preceding code adds the `pdev_simple_dashboard_register()` action to the `wp_dashboard_setup` hook. It then registers a new dashboard widget via the `wp_add_dashboard_widget()` function. The first parameter is a unique ID for the widget and set to `pdev-simple-dashboard`. It is then given a title of “Plugin: Report Bugs.” The third parameter sets the `pdev_simple_dashboard_display()` function as the callback for displaying the actual widget.

Now that your dashboard widget is registered, you need to set up the custom function to display a message to your users.

```
// Dashboard widget callback.
function pdev_simple_dashboard_display() {
    echo '<p>Please contact support@example.com to report
bugs.</p>';
}
```

You now have a custom dashboard widget with a simple message displayed to your users, as shown in [Figure 14-4](#). The Dashboard Widget API automatically makes your widget draggable and collapsible and even adds your widget to the Screen Options tab so that users can easily hide it if they choose.



[FIGURE 14-4](#): Custom dashboard widget

Now review the plugin code in its entirety.

```
<?php
/**
 * Plugin Name: Simple Dashboard Widget
 * Plugin URI: http://example.com/
 * Description: Example plugin of a simple dashboard widget.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'wp_dashboard_setup',
'pdev_simple_dashboard_register' );

// Registers dashboard widgets.
function pdev_simple_dashboard_register() {

    wp_add_dashboard_widget(
        'pdev-simple-dashboard',
        'Plugin: Report Bugs',
        'pdev_simple_dashboard_display'

}

// Dashboard widget callback.
function pdev_simple_dashboard_display() {
    echo '<p>Please contact support@example.com to report
bugs.</p>';
}
```

Creating a Dashboard Widget with Options

Now that you understand dashboard widgets, you can create a more advanced widget that stores an option value. Dashboard widgets can store options, making them easily customizable by the user. If a dashboard widget has any options, a Configure link is displayed when you hover over the widget title.

The dashboard widget in this example enables you to set a custom RSS feed URL and display the contents of that feed.

```
<?php
add_action( 'wp_dashboard_setup',
'pdev_advanced_dashboard_register' );

// Registers dashboard widgets.
```

```

function pdev_advanced_dashboard_register() {

    wp_add_dashboard_widget(
        'pdev-advanced-dashboard',
        'Custom Feed',
        'pdev_advanced_dashboard_display',
        'pdev_advanced_dashboard_control'
    );
}

```

Notice that the `wp_add_dashboard_widget()` function has the fourth parameter set, in this example `pdev_advanced_dashboard_control()`, which is the control callback. This is the function that displays your widget setting field and saves the data entered as an option for your widget.

Next you need to create the `pdev_advanced_dashboard_display()` function to display the custom RSS feed in your widget.

```

// Dashboard widget callback.
function pdev_advanced_dashboard_display() {

    // Get widget option.
    $feed_url = get_option( 'pdev_dashboard_feed_url' );

    // If no feed, set to the WordPress.org news feed.
    if ( ! $feed_url ) {
        $feed_url = 'https://wordpress.org/news/feed';
    }

    // Open HTML wrapper.
    echo '<div class="rss-widget">';

    // Output RSS feed list.
    wp_widget_rss_output(
        esc_url_raw( $feed_url ),
        [
            'title'      => 'RSS Feed News',
            'items'       => 5,
            'show_summary' => true,
            'show_author'  => false,
            'show_date'    => true
        ]
    );

    // Close HTML wrapper.
    echo '</div>';
}

```

The first two lines retrieve the RSS feed URL, which is stored as an option in the database. [Chapter 3](#) covers plugin settings and options in more detail.

Next, the widget uses the `wp_widget_rss_output()` function to retrieve the RSS feed and display it. This handy little function is great for retrieving and displaying RSS feeds in WordPress. The widget defines the RSS URL, sets the title to RSS Feed News, sets the number of posts to show to 5, and includes a few other options.

Now that you have the widget display, you need to create the control callback function, `pdev_advanced_dashboard_control()`. This function adds the form field to your widget and can also save the value entered by the user.

```
// Dashboard widget control callback.
function pdev_advanced_dashboard_control() {

    // Check if option is set before saving.
    if ( isset( $_POST['pdev_dashboard_feed_url'] ) ) {

        // Update database option.
        update_option(
            'pdev_dashboard_feed_url',
            esc_url_raw(
                $_POST['pdev_dashboard_feed_url']
            )
        }

        // Get widget option.
        $feed_url = get_option( 'pdev_dashboard_feed_url' );
    }

    // Get widget option.
    >
    <p>
        <label>
            RSS Feed URL:
            <input
                type="text"
                name="pdev_dashboard_feed_url"
                class="widefat"
                value="<?php echo esc_url(
                    $feed_url );
                ?>">
        </label>
    </p>
    <?php }
```

The first task the function handles is saving the widget option. You should always check to verify that the `POST` value exists prior to saving it using the `isset()` PHP function. Next you sanitize the value of `$_POST['pdev_dashboard_feed_url']` using the `esc_url_raw()` function before saving the option to the database via `update_option()`. The `esc_url_raw()` function will properly format the URL and make sure it does not have any illegal characters.

Once the widget option is saved, you need to display the widget form field so that your users can enter the RSS feed URL they want displayed. First, retrieve the widget option from the database if it exists so you can display it in the form field. Next create a simple text input named `pdev_dashboard_feed_url`. Notice that the value is set to `$feed_url`, which is storing the RSS feed URL value entered by the user.

You now have a custom dashboard widget that stores a custom RSS feed URL and displays the latest two posts to the user, as shown in [Figure 14-5](#).

Dashboard

Custom Feed

[WordPress Leaders Nominated for CMX Awards](#) January 15, 2020

Two members of the WordPress leadership team were nominated for excellent work in their field in the first ever Community Industry Awards. Andrea Middleton is nominated for Executive Leader of a Community Team and Josepha Haden Chomphosy is nominated for Community Professional of the Year. CMX is one of the largest professional organizations dedicated to [...]

[The Month in WordPress: December 2019](#) January 3, 2020

As 2019 draws to a close and we look ahead to another exciting year let's take a moment to review what the WordPress community achieved in December. WordPress 5.3.1 and 5.3.2 Releases The WordPress 5.3.1 security and maintenance release was announced on December 13. It features 46 fixes and enhancements. This version corrects four security [...]

[WordPress 5.3.2 Maintenance Release](#) December 18, 2019

WordPress 5.3.2 is now available! This maintenance release features 5 fixes and enhancements. WordPress 5.3.2 is a short-cycle maintenance release. The next major release will be version 5.4. You can download WordPress 5.3.2 by clicking the button at the top of this page, or visit your Dashboard → Updates and click Update Now. If you have sites that support [...]

FIGURE 14-5: Custom dashboard widget

Now look at the full source for the custom RSS feed dashboard widget.

```
<?php
/**
 * Plugin Name: Advanced Dashboard Widget
 * Plugin URI: http://example.com/
 * Description: Example plugin of a advanced dashboard
 * widget with options.
 * Author: WROX
 * Author URI: http://wrox.com
 */

add_action( 'wp_dashboard_setup',
'pdev_advanced_dashboard_register' );

// Registers dashboard widgets.
function pdev_advanced_dashboard_register() {
```

```
wp_add_dashboard_widget(
    'pdev-advanced-dashboard',
    'Custom Feed',
    'pdev_advanced_dashboard_display',
    'pdev_advanced_dashboard_control'
);
}

// Dashboard widget callback.
function pdev_advanced_dashboard_display() {

    // Get widget option.
    $feed_url = get_option( 'pdev_dashboard_feed_url' );

    // If no feed, set to the WordPress.org news feed.
    if ( ! $feed_url ) {
        $feed_url = 'https://wordpress.org/news/feed';
    }

    // Open HTML wrapper.
    echo '<div class="rss-widget">';

    // Output RSS feed list.
    wp_widget_rss_output(
        esc_url_raw( $feed_url ),
        [
            'title'      => 'RSS Feed News',
            'items'       => 5,
            'show_summary' => true,
            'show_author'  => false,
            'show_date'    => true
        ]
    );
    // Close HTML wrapper.
    echo '</div>';
}

// Dashboard widget control callback.
function pdev_advanced_dashboard_control() {

    // Check if option is set before saving.
    if ( isset( $_POST['pdev_dashboard_feed_url'] ) ) {

        // Update database option.
        update_option(
            'pdev_dashboard_feed_url',
            esc_url_raw(

```

```

        $_POST['pdev_dashboard_feed_url'] )
            );
    }

    // Get widget option.
    $feed_url = get_option( 'pdev_dashboard_feed_url' ); ?
>

<p>
    <label>
        RSS Feed URL:
        <input
            type="text"
            name="pdev_dashboard_feed_url"
            class="widefat"
            value="<?php echo esc_url(
$feed_url ); ?>"
            />
    </label>
</p>
<?php }

```

REWRITE RULES

The Rewrite API is often considered one of the trickiest areas in WordPress and is certainly one of the least documented on the web. This section first gives you some background information on why URLs are rewritten, then explains how to do this in WordPress, and finally shows you a couple real-world scenarios in which you can leverage the Rewrite API.

Unfortunately, given the restraints of publishing, it is impossible to provide a full deep-dive into all of the possibilities with the Rewrite API. Instead, consider this section more of a high-level overview to start you down the path of custom rewrite rules.

Why Rewrite URLs

Dynamic sites use URLs that generate content from query string parameters. These URLs are often rewritten to resemble URLs for static pages on a site with a subdirectory hierarchy. For example, the URL to a wiki page might be <http://example.com/index.php?title=rewrite-url> and be actually rewritten to <http://example.com/rewrite-url>. A request

to this latter, prettier URL will be transparently rewritten by the web server to the former URL.

This introductory section familiarizes you with the concept of “pretty permalinks” (sometimes called “fancy URLs” in web applications) and URL rewriting in general and specifically in WordPress.

Permalink Principles

Web applications and sites can have two completely different audiences: human readers and search engines. Online resources should be friendly to both.

Search Engine Friendly

Suppose you have coded a complete online store for a client, with various products and categories. From a programmer's perspective, each URL of the site would be similar to <http://example.com/shop.php?action=display&category=12&subcat=4>. This URL easily maps to variables that can then typically fetch information from a database or perform actions.

The problem with that URL is that search engines may index it or just index <http://example.com/shop.php>, which may not even return something intelligible.

User Friendly

As a human user, you want a URL to be easy to understand and, if possible, memorable. For instance, consider these two URLs that would actually display the same product page:

- example.com/shop.php?action=display&category=123&product_id=42
- [example.com/shop/shirts/wordpress-blue /](http://example.com/shop/shirts/wordpress-blue/)

The first URL is long and cryptic, whereas the second one is shorter and self-explanatory.

Even when URLs are not obscure like the first one, having a directory-like structure makes it much more understandable. The following two URLs would, for instance, point to the same page on a WordPress-powered site.

- <http://example.com/index.php?year=2020&paged=3>
- <http://example.com/2020/page/3/>

Apache's mod_rewrite

Web server developers have imagined ways to rewrite URLs, from something programmatically convenient (`shop.php?product=100`) to something user and search engine friendly (`/buy/wordpress-shirt/`). This section highlights how this is done with the Apache web server, but other web server software (Lighttpd, Nginx, IIS, and so on) all have similar techniques.

The key module for permalinks in Apache is `mod_rewrite`, a module that enables defining rewrite rules typically found in the `.htaccess` file. A classic rewrite rule consists of the following code block:

```
<IfModule mod_rewrite.c>
  RewriteEngine on
  RewriteRule [ pattern ] [ substitution ] [ optional flag(s) ]
</IfModule>
```

The pattern and substitution parameters can use regular expressions.

Consider the following rewrite rule:

```
RewriteRule /buy/([/]+)/ /shop.php?product=$1 [L]
```

Now, when a client requests a URL that starts with `/buy/` followed several times by a character that is not a slash (`[/]+`) and then a slash, the web server internally redirects the request to `/shop.php` and passes the parameter `product` with the value caught inside the parentheses of the pattern.

If you want to learn more about `mod_rewrite` and URL rewriting in a non-WordPress environment, you can read a thorough guide at <https://www.sitepoint.com/guide-url-rewriting/>.

URL Rewriting in WordPress

A typical WordPress URL such as `/2020/03/hello-world/` doesn't match an actual physical path on the web server. (No “hello-world” directory is in a “03” directory, which is in a “2020” folder.) At some point, the URL was rewritten.

NOTE *If your WordPress setup runs on a capable server (usually Apache with mod_rewrite), you can enable “pretty permalinks” to benefit from the user and search engine-friendly URLs, such as example.com/2020/03/hello-world/ instead of example.com/index.php?p=1. Read more about this beginner feature at <https://wordpress.org/support/article/using-permalinks/>.*

When installed, WordPress creates an `.htaccess` file in its root directory that contains the following block:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^index\.php$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /index.php [L]
</IfModule>
```

This `mod_rewrite` directive contains a conditional rewrite rule, which tells the web server the following:

- If the request is `index.php`, redirect to this file and don't try to match any other rewrite rule. (The `[L]` flag stands for Last.)
- Check whether the request is not a file (`%{REQUEST_FILENAME} !-f`) and not a directory (`%{REQUEST_FILENAME} !-d`).
- Then rewrite the URL to `index.php` and don't try to apply another rewrite rule.

This `.htaccess` directive redirects requests such as `/2020/page/6/` to `/index.php`. This means that practically all requests in the front-end area of a WordPress site are internally redirected to `index.php`, which then has to guess how to interpret the request. Enter the Rewrite API.

How WordPress Handles Queries

You now know that when a visitor loads a WordPress-powered page, the file actually loaded is `index.php`. For example, if you request <http://example.com/2020/01/hello-world/>, WordPress internally redirects this URL into <http://example.com/index.php?p=1> and eventually fetches from the database data for the post with ID 1. How is the translation from a URL to a MySQL query done?

The following section explains what magic happens when the web server displays a WordPress-powered page and how plugins can interfere with this process.

Overview of the Query Process

You need to understand how and when events occur within WordPress because this can highlight the parts where your plugin can interfere with the process. You can now dissect the flow of events when a page is requested and determine which files are included and which functions are called.

1. The root `index.php` file is loaded, as per the `.htaccess` rewrite rule, and loads the file `wp-blog-header.php`.
2. This file loads `wp-load.php`, which searches and includes `wp-config.php`, which will in turn load `wp-settings.php`, which includes the function files, active plugins, and then pluggable functions.
3. Two new objects are instantiated: `$wp_query` and `$wp_rewrite`. You learn about these later.
4. A few other files are loaded, such as translation files and the theme's functions file.

WordPress is now fully loaded and plugins can start interacting, but it doesn't yet know what to display and what page has been requested. Let's get back to `wp-blog-header.php`. After everything is loaded, this file calls the function `wp()`, which starts the magic—the method `WP::parse_request()`.

The `parse_request()` method from the `WP` class prepares everything WordPress needs to know to understand the page request.

5. This function fetches the list of all the registered rewrite rules. Just as previously explained with `mod_rewrite`, it consists of a list of pattern/replacement pairs, to tell WordPress that `/category/tutorials/page/2/` actually means `/index.php?category_name=tutorials&paged=2`.
6. The function goes through each rewrite rule, compares it to the requested URL, and tries to find a match. If no match is eventually found, this is a 404 error.

At this point, if the page is not a 404 error, WordPress now has a permalink translation pattern with query variable placeholders, such as `index.php?category_name=<string>&paged=<number>`. It now needs to get the values of these query variables.

7. The function `parse_request()` now obtains the list of the registered query variables. For each variable, it checks whether a value has been set by the permalink pattern, by POST, or by GET submission.
8. Now, WordPress knows everything it needs to convert the initial URL request into a proper MySQL database query, get post data, load the required theme template, and display the requested page.

Two expressions may have caught your attention in the preceding flow: “registered rewrite rules” and “registered query variables.” If at some point these are registered, then there must be a way for plugins to change things.

The Rewrite Object

The `$wp_rewrite` object is the first object you mess with when playing with the Rewrite API. Let's take a look at its content. Using a simple `print_r($GLOBALS['wp_rewrite'])` displays the following information:

```
WP_Rewrite Object (
  ...
  [permalink_structure] => /%year%/%postname%
  [use_trailing_slashes] => 1
  ...
  [rules] => Array (
    [category/(.+?)/?$_] => index.php?category_name=$matches[1]
    [tag/([^\/]+)/page/?( [0-9]{1,} )/?$_] => index.php?tag=$matches[1]&paged=$matches[2]
  )
)
```

```

[ tag/([^\/]+)/*]
  => index.php?tag=$matches[1]
[(.+?)/trackback/*]
  => index.php?pagename=$matches[1]&tb=1
  ...
)
[endpoints] => Array ()
  ...
)

```

Some of the preceding properties should already be familiar to you. The rules array contains the list of all registered rewrite rules. The \$rewrite object contains all the information related to the permalink structure of your site, such as the complete set of rewrite rules that were fetched at the previous flow or the list of registered feeds and their URL structure.

The Query Object

Similarly and before you learn how to alter it, take an inside look at the global \$wp_query object, with a print_r() call when requesting the page /2020/01/hello-world/ on a WordPress-powered site.

```

WP_Query Object (
  [query_vars] => Array (
    [page] => 0
    [year] => 2020
    [month] => 01
    [name] => hello-world
    ...
  )
  ...
  [is_single] => 1
  [is_preview] =>
  [is_page] =>
  ...
  [query] => Array (
    [year] => 2020
    [name] => hello-world
  )
  ...
)

```

The \$wp_query object defines the list of authorized query variables that can be matched by the rewrite rules and collects all the information needed to translate the initial page request into a MySQL query.

What Plugins Can Do

Using functions of the Rewrite API, plugins can interfere with the `$wp_rewrite` and `$wp_query` objects, for instance to perform the following actions as you will learn in the next section, “Practical Uses”:

- Create your own rewrite rules and define how WordPress will interpret them
- Integrate a WordPress site with non-WordPress pages and keep a consistent URL pattern and site layout
- Create a custom feed with a custom feed permalink

Now that you know the underlying concepts of the Rewrite API, it's time for you to write actual code.

Practical Uses

You will now dive into some practical examples and code for real-world scenarios. In this section, you will learn to do the following:

- Leverage the Rewrite API to easily generate an arbitrary number of subpages under the hierarchy of one parent page
- Define a custom permalink structure to easily integrate non-WordPress content into a WordPress-powered site

Rewriting a URL to Create a List of Shops

You've just redesigned the site of your latest client, a big retail company with dozens of stores across the country. You now have to list these stores within the website. You could manually create a separate page for each store, which could take untold hours. Or, you can create one page at <http://example.com/shops/> and automatically make WordPress understand that <http://example.com/shops/city-name> needs to display the information page for the store located in that city. The latter option is much cleaner.

The function that creates a new rewrite rule is `add_rewrite_rule()`, which needs three arguments: a string defining the URL pattern to be matched,

another string for the URL replacement, and where to place the rule relative to other rules (top or bottom). In your Rewrite Rules Shop plugin, write the following:

```
<?php
// Add rewrite rules.
add_action( 'init', 'pdev_list_stores_add_rules' );

function pdev_list_stores_add_rules() {

    add_rewrite_rule(
        'stores/?(^/)*',
        'index.php?
pagename=stores&store_id=$matches[1]',
        'top'
    );
}
```

This internally redirects all requests to the URL `stores/something/` to the page Stores with an additional parameter, that is, `index.php?pagename=stores&store_id=something`.

Now you need to add the new `store_id` parameter to the list of registered query variables.

```
<?php
// Add the store_id query var so that WP recognizes it.
add_filter( 'query_vars', 'pdev_list_stores_query_var' );

function pdev_list_stores_query_var( $vars ) {
    $vars[] = 'store_id';
    return $vars;
}
```

So far, you have modified the list of defined rewrite rules held in the `$wp_rewrite` object and the list of authorized query variables, kept in the `$wp_query` object. You're almost there!

The trick with rewrite rules is that when they are modified, you need to tell WordPress to refresh and rebuild the list. To do this, you can either visit the Permalink Settings screen in the admin area or use the function `flush_rewrite_rules()`. You can do this on plugin activation and plugin deactivation.

```

// Add the rewrite rule and flush on activation.
add_action( __FILE__, 'pdev_list_stores_activate' );

function pdev_list_stores_activate() {
    pdev_list_stores_add_rules();
    flush_rewrite_rules();
}

// Flush rewrite rules on deactivation.
register_deactivation_hook( __FILE__,
    'pdev_list_stores_deactivate' );

function pdev_list_stores_deactivate() {
    flush_rewrite_rules();
}

```

Note these best practices when adding new rewrite rules:

- On plugin activation, add the rule and flush.
- On `init`, also add the rule, in case another plugin flushes the rules.
- Don't flush rules on every page request (for example, hooking in `init`); that would cause unnecessary overhead.
- On plugin deactivation, flush the rules again to clear the list.

Now review the fully functional plugin code, shown here:

```

<?php
/**
 * Plugin Name: Rewrite Rules Shop
 * Plugin URI: http://example.com/
 * Description: Adds a rewrite rule to list stores as
children of the Stores page.
 * Author:      WROX
 * Author URI: http://wrox.com
 */

// Add the rewrite rule and flush on activation.
add_action( __FILE__, 'pdev_list_stores_activate' );

function pdev_list_stores_activate() {
    pdev_list_stores_add_rules();
    flush_rewrite_rules();
}

```

```

// Flush rewrite rules on deactivation.
register_deactivation_hook( __FILE__,
'pdev_list_stores_deactivate' );

function pdev_list_stores_deactivate() {
    flush_rewrite_rules();
}

// Add rewrite rules.
add_action( 'init', 'pdev_list_stores_add_rules' );

function pdev_list_stores_add_rules() {

    add_rewrite_rule(
        'stores/?(^/)*',
        'index.php?',
        'pagename=stores&store_id=$matches[1]',
        'top'
    );
}

// Add the store_id query var so that WP recognizes it.
add_filter( 'query_vars', 'pdev_list_stores_query_var' );

function pdev_list_stores_query_var( $vars ) {
    $vars[] = 'store_id';
    return $vars;
}

```

That's it for the plugin part. Now

<http://example.com/stores/greenville/> redirects to the Stores WordPress parent page with the additional internal parameter `store_id`.

How this is displayed on the front end will be determined by the active WordPress theme's templates. Theme development is outside the scope of this book. You can learn more about theme template development at <https://developer.wordpress.org/themes/basics/template-files/>.

Creating a New Permalink Structure and Integrating Non-WordPress Pages

In the previous plugin, you created a WordPress page to handle URLs rewritten to it. You can now create other rewrite rules using a different approach.

The client you're working with already has a product listing script and wants you to integrate it in its new website you created for them. You can make WordPress handle all requests to

<http://example.com/shop/something/> and use the existing listing script.

On the Permalink Settings screen in the WordPress admin, you can define custom permalinks using tags such as `%year%` or `%monthnum%`. You will now define a new `%product%` tag and use it in the permalink structure of the site.

```
// Add rewrite rules.
add_action( 'init', 'pdev_product_permalinks_add_rules' );

function pdev_product_permalinks_add_rules() {

    // Add rewrite tag.
    add_rewrite_tag( '%product%', '([^\]+)' );

    // Add permastuct.
    add_permastruct( 'product', 'shop/%product%' );
}
```

The first function call, `add_rewrite_tag()`, defines the tag and what can match it. Here, the tag `%product%` matches one or more characters that are not a forward slash, using the regular expression `([^\]+)`. This function call also registers a new query variable with the same name `product`.

The `add_permastruct()` function describes a new permalink structure. It accepts two parameters: an arbitrary name for the structure and how URLs and tags should be formed.

Now look at the rewrite rules that have been added to the `$wp_rewrite` object and its `rules` property:

```
[shop/([^\]+)/feed/(feed|rdf|rss|rss2|atom)/?$]
  => index.php?product=$matches[1]&feed=$matches[2]
[shop/([^\+)/)(feed|rdf|rss|rss2|atom)/?$]
  => index.php?product=$matches[1]&feed=$matches[2]
[shop/([^\+)/page/?([0-9]{1,})/?$]
  => index.php?product=$matches[1]&paged=$matches[2]
[shop/([^\+)/?$]
  => index.php?product=$matches[1]
```

Example URLs matching these rewrite rules could be the following.

- <http://example.com/shop/something/feed/rss2/>
- <http://example.com/shop/stuff/atom/>
- <http://example.com/shop/thing/page/3/>
- <http://example.com/shop/item/>

These URLs can internally redirect to the following:

- <http://example.com/index.php?product=something&feed=rss2>
- <http://example.com/index.php?product=stuff&feed=atom>
- <http://example.com/index.php?product=thing&paged=3>
- <http://example.com/index.php?product=item>

Using just two function calls, you created a completely new permalink structure that can handle pagination and feed generation!

Now that requests to /shop/something/ successfully redirect to /index.php?product=something, you can integrate the existing product listing script. In the following code, the actual script integration is commented out and replaced with simple output on the front end.

```
// Output product script on front end.
add_action( 'template_redirect', 'pdev_products_display' );

function pdev_products_display() {
    if ( $product = get_query_var( 'product' ) ) {

        // Includes the client's product script.
        // include display-product.php;

        printf(
            'Searching for product: %s?',
            esc_html( $product )
        );
        exit();
    }
}
```

By hooking into the early action 'template_redirect', you can hijack the normal page display and, if the product query variable is set, include the

shop listing script. Don't forget to use `exit()` so that WordPress does not try to further handle the page display and output a 404 error.

To test for pagination or feed generation, you can also check the values of `get_query_var('paged')` and `get_query_var('feed')`.

You must also flush the rewrite rules on activation and deactivation. Then, you will be done with the plugin.

```
// Add the rewrite rule and flush on activation.
add_action( __FILE__, 'pdev_product_permalinks_activate' );

function pdev_product_permalinks_activate() {
    pdev_product_permalinks_rules();
    flush_rewrite_rules();
}

// Flush rewrite rules on deactivation.
register_deactivation_hook( __FILE__,
    'pdev_product_permalinks_deactivate' );

function pdev_product_permalinks_deactivate() {
    flush_rewrite_rules();
}
```

Now review the full plugin code.

```
<?php
/**
 * Plugin Name: Product Permalinks
 * Plugin URI: http://example.com/
 * Description: Adds custom product rewrite tag and
permastruct.
 * Author:      WROX
 * Author URI:  http://wrox.com
 */

// Add the rewrite rule and flush on activation.
add_action( __FILE__, 'pdev_product_permalinks_activate' );

function pdev_product_permalinks_activate() {
    pdev_product_permalinks_rules();
    flush_rewrite_rules();
}

// Flush rewrite rules on deactivation.
register_deactivation_hook( __FILE__,
```

```

'pdev_product_permalinks_deactivate' );

function pdev_product_permalinks_deactivate() {
    flush_rewrite_rules();
}

// Add rewrite rules.
add_action( 'init', 'pdev_product_permalinks_add_rules' );

function pdev_product_permalinks_add_rules() {

    // Add rewrite tag.
    add_rewrite_tag( '%product%', '([^\]+)' );

    // Add permastuct.
    add permastuct( 'product', 'shop/%product%' );
}

// Output product script on front end.
add_action( 'template_redirect', 'pdev_products_display' );

function pdev_products_display() {

    if ( $product = get_query_var( 'product' ) ) {

        // Includes the client's product script.
        // include display-product.php;

        printf(
            'Searching for product: %s?',
            esc_html( $product )
        );
        exit();
    }
}

```

THE HEARTBEAT API

The Heartbeat API is a little-used, basic server polling API included in WordPress. It allows plugins to send, receive, and process data in almost real time.

What Is the Heartbeat API?

When a page is loaded, the Heartbeat API creates an interval (also called a *pulse* or *tick*). By default, this interval runs every 15 seconds. Like a human heartbeat, the Heartbeat API ticks every so often, albeit a little slower.

The tick provides an opportunity for plugins to latch onto the API and run custom code. Because the heartbeat is running practically in real time, it is almost entirely served via client-side JavaScript.

Using the Heartbeat API

When using the Heartbeat API, plugins can attach events on the client side (JavaScript) for sending data and processing the response from the server. On the server side, plugins can receive data and send a response to the client side.

Sending Data

On the `heartbeat-send` event, you can use JavaScript to send any data back to the server that you want. This can come from user input or anywhere. Take a look at the following code snippet that generates a random number and sends it:

```
jQuery( document ).on( 'heartbeat-send', function ( event, data ) {  
    // Send a random number to the server.  
    data.pdev_random_number = Math.random();  
});
```

To pass custom data to the server, you simply need to add a new item to the `data` object, which is the second parameter for your callback function. In the preceding code, you generated a random number and added it via `data.pdev_random_number`.

Receiving and Responding to Data

On the server side, you have the option of handling the data. You can skip this step if you do not need to run any server-side code to work with the data. However, this can be useful for sanitizing, validating, or even saving data to the database.

For your random number generated via JavaScript, you now add a filter on the heartbeat_received hook. The filter will sanitize the data and store it in the database.

```
add_filter( 'heartbeat_received',
'pdev_random_number_received', 10, 2 );

function pdev_random_number_received( $response, $data ) {

    // Bail if no plugin data sent.
    if ( empty( $data['pdev_random_number'] ) ) {
        return $response;
    }

    // Sanitize the data.
    $response['pdev_random_number'] = floatval(
$data['pdev_random_number'] );

    // Update database option:
    update_option(
        'pdev_heartbeat_random_number',
        $response['pdev_random_number']
    );

    // Return the response.
    return $response;
}
```

WARNING *Complex calculations or other heavy scripts can slow down a server. Be sure to use as few resources as possible because the Heartbeat API can pulse many times while someone is on the website. Just be mindful of the resources you use.*

Processing the Response

Once the data is sent back to the server, it gets sent back to the client side. From this point, you can do anything you want via JavaScript. This could be something as simple as updating onscreen data, logging the response, or creating an alert popup.

Now create a simple alert popup, attached to the heartbeat-tick event.

```

jQuery( document ).on( 'heartbeat-tick', function ( event,
data ) {

    // If no data, bail.
    if ( ! data.pdev_random_number ) {
        return;
    }

    alert( 'The random number is ' +
data.pdev_random_number );
} );

```

The previous code first checks whether your plugin's data was sent. If so, it uses JavaScript's built-in `alert()` to display a message.

Full Heartbeat API Plugin

Now, you will put all of the preceding code into a complete plugin. Create a new `plugin-heartbeat-random-number` folder to house your PHP and JavaScript files.

Then, create a new file named `plugin.php` with the following code. Note that it loads the `heartbeat.js` file on the `wp_enqueue_scripts` hook, which is the front-end hook for loading JavaScript files.

```

<?php
/**
 * Plugin Name: Heartbeat Random Number
 * Plugin URI: http://example.com/
 * Description: Generates and saves a random number using
the Heartbeat API.
 * Author: WROX
 * Author URI: http://wrox.com
 */

// Load heartbeat.js file.
add_action( 'wp_enqueue_scripts',
'pdev_random_number_scripts' );

function pdev_random_number_scripts() {

    wp_enqueue_script(
        'pdev-heartbeat-random-number',
        plugin_dir_url( __FILE__ ) . 'heartbeat.js',
        [ 'heartbeat', 'jquery' ],

```

```

        null,
        true
    );
}

// Receive data from heartbeat.
add_filter( 'heartbeat_received',
'pdev_random_number_received', 10, 2 );

function pdev_random_number_received( $response, $data ) {

    // Bail if no plugin data sent.
    if ( empty( $data['pdev_random_number'] ) ) {
        return $response;
    }

    // Sanitize the data.
    $response['pdev_random_number'] = floatval(
$data['pdev_random_number'] );

    // Update database option.
    update_option(
        'pdev_heartbeat_random_number',
        $response['pdev_random_number']
    );

    // Return the response.
    return $response;
}

```

Finally, create a `heartbeat.js` file within the plugin folder with the following code:

```

jQuery( document ).on( 'heartbeat-send', function ( event,
data ) {

    // Send a random number to the server.
    data(pdev_random_number = Math.random());
} );

jQuery( document ).on( 'heartbeat-tick', function ( event,
data ) {

    // If no data, bail.
    if ( ! data(pdev_random_number) ) {
        return;
    }
}

```

```
    alert( 'The random number is ' +
data.pdev_random_number );
} );
```

SUMMARY

We named this chapter “The Kitchen Sink” for a reason. It contains a lot of information on several different subjects that didn't have a more appropriate home in another chapter or were not important enough for their own chapter with how WordPress plugins are built today. Some of the items like widgets and shortcodes, especially shortcodes, could be obsolete in a year or two. Other topics like the Rewrite API and the Heartbeat API are useful but not used by many plugins. And, subjects like querying posts and building dashboard widgets are still useful today but didn't warrant full chapters.

Regardless of the topic, you should now have a good high-level overview of each item to carry you forward in your plugin development journey.

15

Debugging

WHAT'S IN THIS CHAPTER?

- Understanding compatibility
- Supporting many WordPress versions
- Playing nicely with other WordPress plugins
- Keeping updated with WordPress
- Debugging your plugins
- Logging debug errors
- Using the Query Monitor plugin

CODE DOWNLOADS FOR THIS CHAPTER

The code downloads for this chapter are at
www.wiley.com/go/prowordpressdev2e on the Downloads tab.

WordPress provides many different ways to help you quickly identify problems with your code. Throughout this chapter, you will learn how to correctly support many different versions of WordPress, how to debug issues with different plugins active, and how to step through code live during runtime. The topics presented in this chapter are a few of the basic things that you are expected to understand to make sure your plugins are bug free, optimized, and ready for whatever WordPress throws at them.

COMPATIBILITY

As a professional WordPress plugin developer, you need to make sure the plugins you create are compatible with whatever kind of WordPress installation they get activated on. This sounds simple enough, but the possibilities and combinations are truly endless.

Supporting Many WordPress Versions

While the latest and greatest versions of WordPress will always provide the best APIs and features, you may not always have the luxury of basing your plugins on it exclusively. In fact, it's more likely you will have no control over what version of WordPress the users of your plugins are on. Their version could be years old, they might be testing the latest nightly build, and your plugin needs to understand the differences to avoid your plugin not working or, even worse, breaking their site completely.

The WordPress project is hugely committed to maintaining backward compatibility, sometimes going to great lengths and huge efforts to avoid breaking changes between releases. This creates a much more developer-friendly environment than if things were breaking all the time, because you can always count on the code that you rely on to continue to do what it always has (even when a new version of WordPress is released) but also means you will have some hard decisions to make for your own plugins.

New major versions of WordPress are released about three times per year, but that can change based on leadership, goals, roadmap, and so on. A good rule of thumb for your own plugins is to maintain backward compatibility for somewhere between three and six major WordPress release cycles. This way, you are able to stay current with the latest WordPress features, while also giving all of your users a few years to get updated.

Some plugins you create will always be backward compatible because they do not use any of the newest functionality in WordPress, and other more complex plugins (like WooCommerce or BuddyPress) support many older versions because they still have hundreds of thousands of people using them.

PHP offers a few different helper functions you should use if you are planning on relying on and using the latest WordPress code available. You can tell what version of WordPress something was introduced in by looking at the documentation block for the `@since` tag. Here's what the `block_version()` function looks like:

```
/**  
 * Returns the current version of the block format that the  
 * content string is using.  
 */
```

```

* If the string doesn't contain blocks, it returns 0.
*
* @since 5.0.0
*
* @param string $content to test.
* @return int The block format version is 1 if the content
contains
    @one or more blocks, 0 otherwise.
*/
function block_version( $content ) {
    return has_blocks( $content ) ? 1 : 0;
}

```

If your plugin is using this function but you need to support both WordPress 4.9 and 5.0, you could use the PHP function `function_exists()` to check whether the user is using a version of WordPress that includes that function in it.

```

<?php

// Block Version is available
if ( function_exists( 'block_version' ) ) {
    $version = block_version( $content );

// Block Version unavailable
} else {
    $version = 0;
}
?>

```

The code snippet uses `function_exists()` to check whether the `block_version()` function exists. If it does exist, this means the user is using WordPress 5.0 or newer, and the plugin can use the `block_version()` function. If the function does not exist, the plugin can fall back to an alternative way to handle the functionality.

PHP also offers `class_exists()` and `method_exists()` to check whether a class or class method exists, and it offers `is_callable()` to verify that the contents of a variable can be called as a function. You will find yourself using these helpers more and more as you begin to support a wider array of WordPress releases.

Similarly, WordPress comes with dozens of its own helper functions designed to be used under specific circumstances. Some are environmental,

while others help with validation or formatting. These functions will typically start with either `is_` or `wp_is_`, but there are a few outliers.

Playing Nicely with Other WordPress Plugins

As a professional WordPress plugin developer, the code you write in your plugins will be running alongside any number of other plugins, many of which are probably not ones you authored yourself. At the time of this writing, there are more than 80,000 plugins available for free in the WordPress Plugin Directory and tens of thousands that are privately given away or sold outside it.

It is impossible for you to know what all of these plugins do or to predict which ones may be activated at the same time as yours in the future, so you will need to code defensively to avoid conflicts and collisions, which you will inevitably run into, either in your own code or in someone else's.

WordPress includes plugins in such a way that they are executed immediately inline and side by side with WordPress itself. Located in `wp-settings.php`, here is the exact code that loops through active and valid plugins and includes them:

```
// Load active plugins.
foreach ( wp_get_active_and_valid_plugins() as $plugin ) {
    wp_register_plugin_realpath( $plugin );
    include_once( $plugin );

    /**
     * Fires once a single activated plugin has loaded.
     *
     * @since 5.1.0
     *
     * @param string $plugin Full path to the plugin's
     * main file.
     */
    do_action( 'plugin_loaded', $plugin );
}
unset( $plugin );
```

The `include_once()` line in the preceding code ensures that a plugin is only ever included one single time. This is another common way to avoid fatal errors in WordPress, as including the same file multiple times would also redefine its classes and functions, leading to a broken WordPress

installation. It is slower than its partner `include()` but is generally considered safer in WordPress due to how unpredictable including files can be.

When WordPress includes a main plugin file, the code inside that file is interpreted and executed immediately, which makes it the perfect place to do some environmental checking and preliminary setup (aka bootstrapping) but a bad place to perform much else. This is because plugins are included relatively early inside WordPress (before much of WordPress has had a chance to get set up itself), so most of what you may be hoping to modify will not even exist yet.

The way WordPress loads plugins also means that every plugin is included in the global scope. This means that every plugin has the ability to change WordPress in any way necessary, and plugins other than yours will be doing so. Everything being loaded up in the same global scope means you will want to be consistent with the following globally scoped concepts:

- Function names
- Class names
- Namespaces
- Action hooks
- Filter hooks
- Global variables

It will also help you immensely to become comfortable exploring the source code of other WordPress plugins. You will pick up on their ideas, patterns, methodologies, and approaches to solving problems similar to those you will face, which in turn will help you create plugins that play nicely with other plugins.

Some larger plugins are even designed to be just as extensible as WordPress is itself, allowing for plugin developers to extend them, essentially writing plugins for plugins. WooCommerce, Easy Digital Downloads, BuddyPress, and bbPress are just a few examples of massive plugins with thriving add-on and third-party communities, where developers are writing WordPress plugins that rely on or integrate specifically with another plugin.

Whether you are writing a stand-alone plugin or one that seeks to extend an existing one, it is inevitable that someone will identify a conflict or collision between their plugin and one of yours. When this happens (and it will happen), you will find the majority of other plugin developers to be kind, helpful, and willing to share their knowledge for the betterment of your mutual WordPress users. It may be your fault or theirs, but that doesn't matter—what matters is improving the code so that users have the best possible experience regardless of what plugins they have installed.

Keeping Current with WordPress Development

One of the main reasons why WordPress is the most popular piece of software for publishing to the web is that it acts as its own compatibility layer between all of the software running on the server and the user who is actually using WordPress to write and publish their content. This means that WordPress is able to adapt to whatever libraries or modules are activated and available in the web server software being used.

Just like WordPress, software interpreters like PHP and web servers like Apache or Nginx can be configured differently from place to place, and WordPress exists to make all of those different configurations completely invisible.

In addition, the project's undying commitment to maintaining backward compatibility means that developers like you can code with confidence that a new WordPress version will not break any of the plugins you've written.

WordPress is constantly changing. Not a single day will go by without a core committer or contributor submitting a patch to fix a bug, inventing a fancy new feature, or proposing different ideas about the direction of WordPress itself. It is a vast and growing community with many moving parts and avenues where ideas are flowing in and out of various stages of implementation.

Even though WordPress changes frequently and rapidly, you will want to do your best to keep up with what is changing and why. In doing so, you will be the first to

- Find new features and functions
- Find out when old functions are deprecated or removed

- See when a WordPress bug affects one of your plugins
- Take advantage of the latest performance improvements

You will find that you can sometimes eliminate many lines of your own plugin code when new functionality is introduced into WordPress. You can remove old functionality and replace it with something new when functions are deprecated. And you can make your own life easier by not shimming work-arounds for WordPress bugs that you have been maintaining all by yourself.

You should also know how releases are handled within WordPress. Understanding this simple concept can enable you to know when you need to do an overview of your plugins to see whether an update is needed. The two types of releases are major releases and minor (or point) releases.

- **Major releases:** 5.4, 5.3, 5.2, and so on. These releases are made available every few months with new features, new functionality, and bug fixes. Many code changes are made between major releases.
- **Point releases:** 5.3.1, 5.3.2, 5.3.3, and so on. These releases are made as needed between major releases and contain bug fixes and security improvements only. New functionality is rarely added in a minor release.

Being involved in the broader WordPress community will help your plugin development in many ways. The most important place to keep track of changes is the Make Network, located at make.wordpress.org. This is a group of blogs (one for each contributor group), general news, and upcoming team meetings and events. The development tracker located at core.trac.wordpress.org is also really important, as it is where the bulk of WordPress code changes occur. It is powered by a piece of software called Trac. It may take a little time to understand how the system works, but when you do, you'll probably even find yourself getting involved in WordPress development. The best way to start learning the system is to simply start reading some tickets and do your best to follow along with the development flow.

Starting with WordPress 5.0, the introduction of a new block-based editor means that some development is starting to happen on GitHub, which is

then later merged into WordPress itself. This is good in that it provides a bit more freedom to people contributing to that part of the source code but bad in that it further fragments and complicates the entire core development process. It is mostly up to you to decide how much value you derive from following along, but today WordPress is all about blocks, so try to follow along if you can.

NOTE *WordPress is constantly changing. Not a single day will go by without a core committer or contributor submitting a patch to fix a bug, inventing a fancy new feature, or proposing different ideas about the direction of WordPress itself.*

Deprecation

When something is deprecated, it is being phased out and should no longer be used. When a function gets deprecated, it is almost always replaced by something newer and better. Sometimes something as small as a function parameter gets deprecated, or other times as large as an entire file gets removed.

Deprecated functions usually live within one of several WordPress files.

- `wp-includes/deprecated.php`: General functions
- `wp-includes/ms-deprecated.php`: Multisite functions
- `wp-includes/pluggable-deprecated.php`: Pluggable functions
- `wp-admin/includes/deprecated.php`: Admin functions

Browsing through the core code for deprecated functions might seem tedious, but it's a good idea to become familiar with them so that you can recognize them, avoid using them, and use their updated equivalents instead.

Dealing with Obsolete Client Installs

When you release a plugin to be available to the general public, you will have no control over whether those users keep their WordPress installation up-to-date, but when you deal directly with your clients, you can

communicate directly with them and maybe even have some administration access to their WordPress installation.

If you work with clients who use an outdated version of WordPress, you should consider it your responsibility to educate them on the benefits of using the latest and most secure version, along with any plugins and themes they are using. If you have a client who does not want to update, here is a handy list of reasons why they should update:

- It makes developing their plugins easier.
- It can cut back on development time and cost.
- Performance improvements may decrease hosting costs.
- There is less chance of their site being targeted and hacked.

The most important thing you can stress is the increased security of each new version. A paying client tends to perk up and pay attention when you can tell them a story of a different site having been vandalized because of an out-of-date WordPress installation.

When upgrading any sites, the most important thing you can do is keep multiple backups of files and the database and make sure you are able to quickly and easily restore from a backup if needed. The WordPress update process is usually simple and painless, but if something goes wrong, you will definitely want to be able to bring everything back online fast.

Many WordPress hosting companies automatically manage backups for you and have convenient interfaces for restoring everything back to normal again.

DEBUGGING

Modern web applications are complicated things. It isn't just your plugins that need debugging, it's also WordPress itself, as well as all of the other plugins and themes that happen to be installed and active at the time something has gone wrong. By following the steps outlined in this section, you will become a master code debugger.

Debugging is the act of finding and removing errors, mistakes, defects, or faults in the relevant code. These kinds of issues are almost exclusively the result of human error, due to either inaccurate logic or a gap in knowledge related to the problem.

PHP is interpreted in real time, and the major benefit of that is you get to see your code changes instantly, without the need to recompile an entire application to do so. This means that something as simple as invalid syntax will immediately crash the entire site, but it also makes finding and fixing something so obvious relatively straightforward.

Whenever you are not working on a live site in a production environment, you should have all of the various debug modes enabled. Put another way, you should never enable any debug code in a live production environment, as doing so is likely to leak sensitive server information that any nefarious visitor to your site could use against you to gain unauthorized access to other parts of the system.

Without requirements or design, programming is the art of adding bugs to an empty text file.

LOUIS SRYGLEY

Enabling Debugging

In this section, you will enable debugging for your entire WordPress installation. You must enable debugging whenever developing plugins. It will make your plugin better, and your job easier, by letting you know what issues arise during the development phase.

At the root of your WordPress installation, open the `wp-config.php` file. This file includes all of the fundamental configuration settings that are required by WordPress to connect to the database, generate secure random numbers, and then continue loading the rest of WordPress.

Near the bottom, you should see the following:

```
/**  
 * For developers: WordPress debugging mode.  
 *  
 * Change this to true to enable the display of notices  
 * during development.
```

```
 * It is strongly recommended that plugin and theme
  developers use WP_DEBUG
  * in their development environments.
  *
  * For information on other constants that can be used for
  debugging,
  * visit the Codex.
  *
  * @link https://codex.wordpress.org/Debugging_in_WordPress
  */
define( 'WP_DEBUG', false );
```

Just like the inline code documentation says, changing the `WP_DEBUG` constant from `false` to `true` will enable WordPress debugging mode. By default, debugging mode is off, and you must manually enable it here by editing this file.

That line of code should now look like this:

```
define( 'WP_DEBUG', true );
```

Enabling debugging mode makes two important things possible.

- Debug messages will now be displayed directly on the screen as they happen. This feature can be selectively toggled off by setting `WP_DEBUG_DISPLAY` to `false`.
- Allows error logging to be enabled, which is covered in the “Error Logging” section of this chapter.

Displaying Debug Output

After you enable debugging mode, refresh your WordPress installation in your preferred web browser. If you're lucky, you may suddenly see error messages displayed in various places on the page where there were none previously.

Now let's take a look at what a deprecated function looks like when used with debugging enabled. Suppose you wanted to get all of the possible category IDs. Prior to WordPress 4.0, the function you would use for this would be `get_all_category_ids()`, but now that this function is deprecated, using it will output a debug notice that reads as follows:

```
Notice: get_all_category_ids is deprecated since version
4.0.0!
Use get_terms() instead.
```

This notice immediately tells you several things about that function.

- The deprecated function name
- The WordPress version where the function was deprecated
- The replacement function to use instead

To update your code, simply swap out the deprecated function with the replacement function. In some instances, there is no replacement, and you will have to deal with those scenarios on a case-by-case basis—each situation may have a different or unique solution.

Making simple changes like this one will bring your code up-to-date and remove any debug messages relating to it from the rest of the site.

Understanding Debug Output

Most debug output is the result of PHP notices, which are related to but less severe than a PHP warning or alert. Even though a notice is relatively innocuous (it will not fatally error a site), it is still important to track them all down and fix them, as they are usually an indication of more severe problems with the surrounding code that could eventually cause worse breakage.

You should consider it your responsibility to always debug your code and to eliminate all notices, warnings, and alerts from your plugins. In this section are several of the most common debug outputs and how to address them.

- Deprecated WordPress functions
- Undefined variable/property notice
- Trying to get the property of a nonobject notice

To see debug notices in action, you must first write a plugin to expose them. This is one of the few times in this book that you will be asked to write

code that is intentionally broken, but what better way is there to learn than by making mistakes?

For this example, we are going to keep things simple. This plugin will attempt to add the author's name and bio at the end of every post. Create a new file named pdev-notice-plugin.php in your /wp-content/plugins/ directory and add the following code to the file:

```
<?php
/**
 * Plugin Name: Debug Notices
 * Plugin URI: https://wrox.com
 * Description: Plugin to create debug errors
 * Version: 0.0.1
 * Author: WROX
 * Author URI: https://wrox.com
 */

add_filter( 'the_content', function( $content = '' ) {

    // If viewing a post with the 'post' post type
    if ( 'post' === $post->post_type ) {
        $author_box = '<div class="author-box">';
        $author_box .= '<h3>' . get_the_author_meta(
            'display_name' ) . '</h3>';
        $author_box .= '<p>' . get_the_author_description()
        . '</p>';
        $author_box .= '</div>';
    }

    // Append the author box to the content.
    $content = $content . $author_box;

    // Return the content.
    return $content;
} );
```

There are several issues with this plugin. With debugging enabled, you should be looking at a screen with three separate debug notices. There is actually a fourth problem with this code, but one is hidden away because the surrounding code is not exposing it yet.

The three notices are as follows:

```
Notice: Undefined variable: post in /wp-
content/plugins(pdev-notice-plugin.php
```

```
on line 14
Notice: Trying to get property 'post_type' of non-object in
/wp-content/plugins/pdev-notice-plugin.php on line
14
Notice: Undefined variable: author_box in /wp-
content/plugins/
pdev-notice-plugin.php on line 22
```

The first and worst issue is with this line:

```
if ( 'post' === $post->post_type ) {
```

PHP is attempting to interpret what the \$post variable is, but because it has not been set or defined in the current scope, an “undefined variable” debug notice is triggered. In addition, because that variable does not exist, it also definitely is not an object, so a “Trying to get property of non-object” debug notice is also triggered. Both of these problems can be fixed a few different ways in WordPress, but one way is better than all of the rest. You can use the `get_post()` function to get whatever the current global \$post variable is, without needing to define or touch the global yourself, like so:

```
$post = get_post();
```

The reason this is the best approach is because defining global variables inside your local function or method scope puts them at risk of accidentally being overwritten. Changing the contents of a global variable almost always has negative consequences and should be avoided whenever possible.

After fixing these two issues, the third debug notice will disappear when viewing a blog post but remain when viewing a page. This notice is caused by the next line of the plugin, resulting in an “undefined variable” notice.

```
$content = $content . $author_box;
```

The problem with this line is that the \$author_box variable will not always be set, so PHP is attempting to concatenate a string with an invalid variable reference. This notice is one of the issues you will run into quite frequently and is one the easiest issues to fix.

It is best practice to either explicitly set up your local variables with intelligent default values near the top of your functions or check whether they have been set before trying to use them.

In this circumstance, you may want to use the PHP function `isset()` to check whether the `$author_box` variable has been set before appending it to the `$content` variable.

```
if ( isset( $author_box ) ) {
    $content = $content . $author_box;
}
```

An even better way to check a variable like this while also avoiding unnecessary processing is with the PHP function `empty()`. The nice thing about `empty()` is that it returns true for all “falsy” values, including null, an empty string, or a zero value. This way, if `$author_box` is set but does not actually contain anything useful, PHP will not need to concatenate anything.

```
if ( ! empty( $author_box ) ) {
    $content = $content . $author_box;
}
```

Now that you've fixed these first three issues, you will see a fourth debug notice appear, letting you know that you used a deprecated function: `get_the_author_description()`. The message also tells you that it has been replaced by `get_the_author_meta('description')`. This debug message is caused by this line of the plugin:

```
$author_box .= '<p>' . get_the_author_description() .
'</p>';
```

Once again, this is a simple problem to solve. You just need to replace the only usage of this deprecated function with the new function being recommended, and the new code will look like the following:

```
$author_box .= '<p>' . get_the_author_meta( 'description' ) .
'</p>';
```

Making these small changes clears the plugin of debug notices completely. It can also make the intended functionality of the plugin work. After completing each change, your code appears as shown in the following plugin:

```
<?php
/**
 * Plugin Name: Debug Notices
```

```

* Plugin URI: https://wrox.com
* Description: Plugin to create debug errors
* Version: 0.0.1
* Author: WROX
* Author URI: https://wrox.com
*/

add_filter( 'the_content', function( $content = '' ) {
    $post = get_post();

    // If viewing a post with the 'post' post type
    if ( ! empty( $post ) && ( 'post' === $post->post_type ) )
    {
        $author_box = '<div class="author-box">';
        $author_box .= '<h3>' . get_the_author_meta(
        'display_name' ) . '</h3>';
        $author_box .= '<p>' . get_the_author_meta(
        'description' ) . '</p>';
        $author_box .= '</div>';
    }

    // Append the author box to the content.
    if ( ! empty( $author_box ) ) {
        $content = $content . $author_box;
    }

    // Return the content.
    return $content;
} );

```

ERROR LOGGING

WordPress offers another useful tool for debugging your site, known as *error logging*. This feature creates a `debug.log` file that continually has any problems appended to it as they happen anywhere within your WordPress installation.

This is super useful for debugging issues with plugins because it logs every error to a file on the server allowing you to easily read through all of the issues at once. This is most useful for live sites, where you never want to display any debug output for visitors to see, but you may still want to keep track of problems in a way that is hidden from public viewing.

Enabling Error Logging

By enabling error logging, you gain an easy way to keep track of all debug output on your site in a central location. It gives you the exact debug notices along with the date and time the error occurred. Each entry is saved at the end of a `debug.log` file, located inside the `wp-content` directory.

To enable error logging, you need to edit the `wp-config.php` file as you did earlier when you turned on `WP_DEBUG` mode. This time, you need to turn on `WP_DEBUG_LOG` by adding the following code to the file:

```
define( 'WP_DEBUG_LOG', true );
```

The debugging section of your `wp-config.php` file should now look like this:

```
define( 'WP_DEBUG', true );
define( 'WP_DEBUG_LOG', true );
```

Together, these two lines tell WordPress to enable debugging mode and that all debug output should be saved to the error log file.

You may want to optionally disable the display of debug messages on the site and save them in the debug file only. This is useful if you want to keep track of debug output on a live site. To do this, you need to add two new constants to your `wp-config.php` file.

```
define( 'WP_DEBUG_DISPLAY', false );
```

Internally, this also tells all of PHP to suppress displaying errors, as follows:

```
ini_set( 'display_errors', 0 );
```

Setting Log File Location

When error logging is enabled, by default WordPress will create a file named `debug.log` in the `wp-content` folder. In most situations, this location and filename will work fine. If you want to change it to something else, you have the option to do so by setting the `WP_DEBUG_LOG` constant to the full path to where the log file should be located.

```
define( 'WP_DEBUG_LOG', '/var/log/wordpress/debug.log' );
```

This example tells WordPress to save the debug log to a specific server path and filename. This path must already exist, and whatever web server user is used for WordPress must also have write permissions to the directory; otherwise, the log will fail to be created or written to.

Understanding the Log File

The `debug.log` file can be read with any basic text editor, so you should not have any trouble opening it and reading its contents.

In the “Debugging” section, you created an error plugin. You also learned how to correct debug messages displayed on the screen. The error log file is another way to view that debug output, and the content that is saved in the log file will look exactly the same as what you saw onscreen. The following is a snippet from the debug log file that the plugin you created earlier will produce:

```
[01-Oct-2019 00:00:00] PHP Notice: Undefined variable: post
in
          /wp-content/plugins/pdev-notice-
plugin.php on line 14

[01-Oct-2019 00:00:00] PHP Notice: Trying to get property
'post_type' of
          non-object in /wp-
content/plugins/pdev-notice-plugin.php
          on line 14

[01-Oct-2019 00:00:00] PHP Notice: Undefined variable:
author_box in /wp-content/
          plugins/pdev-notice-plugin.php on
line 22
```

As you can see, each notice creates a new entry in the file, and each entry looks nearly the same as what it looked like in the “Debugging” section. The only exception/bonus here is that you have an exact date and time of when the output occurred.

After reading your error log entries, you should revisit the “Debugging” section to work through all of the problems that it reveals. The error log entries do not get removed from the file after you have fixed their related issue on the site and in your plugin, so be sure to pay attention to the date

and time stamps so that you are not looking for issues that you have already fixed.

QUERY MONITOR

When it comes to debugging tools for WordPress, the Query Monitor plugin by John Blackbourn is by far the single most important and useful debugging tool at your disposal. It enables debugging of database queries, PHP errors, actions and filters, block editor blocks, enqueued scripts and stylesheets, HTTP API calls, and more.

It includes advanced features such as debugging Ajax calls, REST API calls, user capability checks, and the ability to narrow down much of its output by specific plugin or theme, allowing you to quickly identify poorly performing plugins, themes, or functions.

It focuses heavily on presenting its information in a useful manner, by showing aggregate database queries grouped by the plugins, themes, or functions that are responsible for them. It adds an Admin Toolbar menu item ([Figure 15-1](#)), showing an overview of the current page with complete debugging information shown in panels as you select a specific menu item.

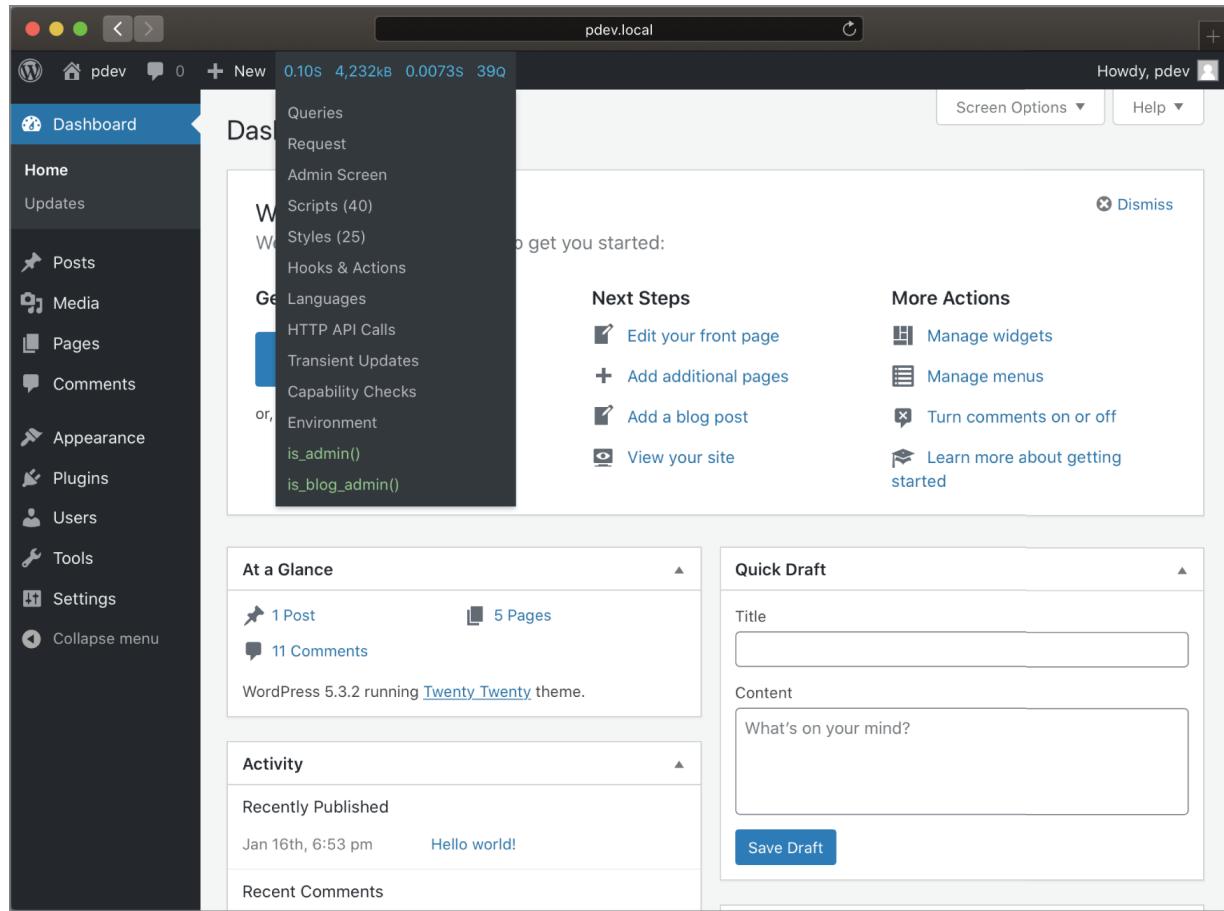
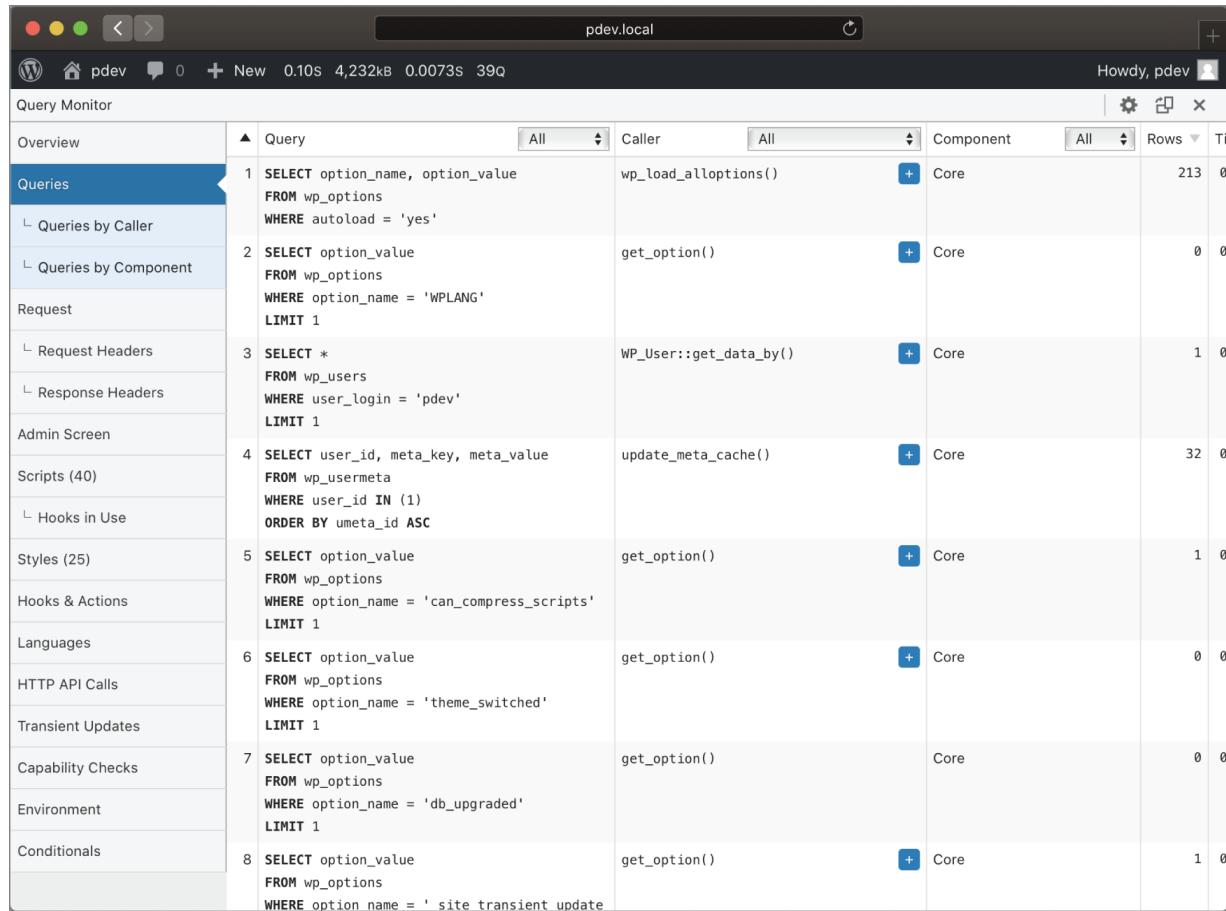


FIGURE 15-1: Admin Toolbar menu item

The drop-down menu provides a bunch of quick links to specific panels in the Query Monitor interface, as shown in [Figure 15-2](#).



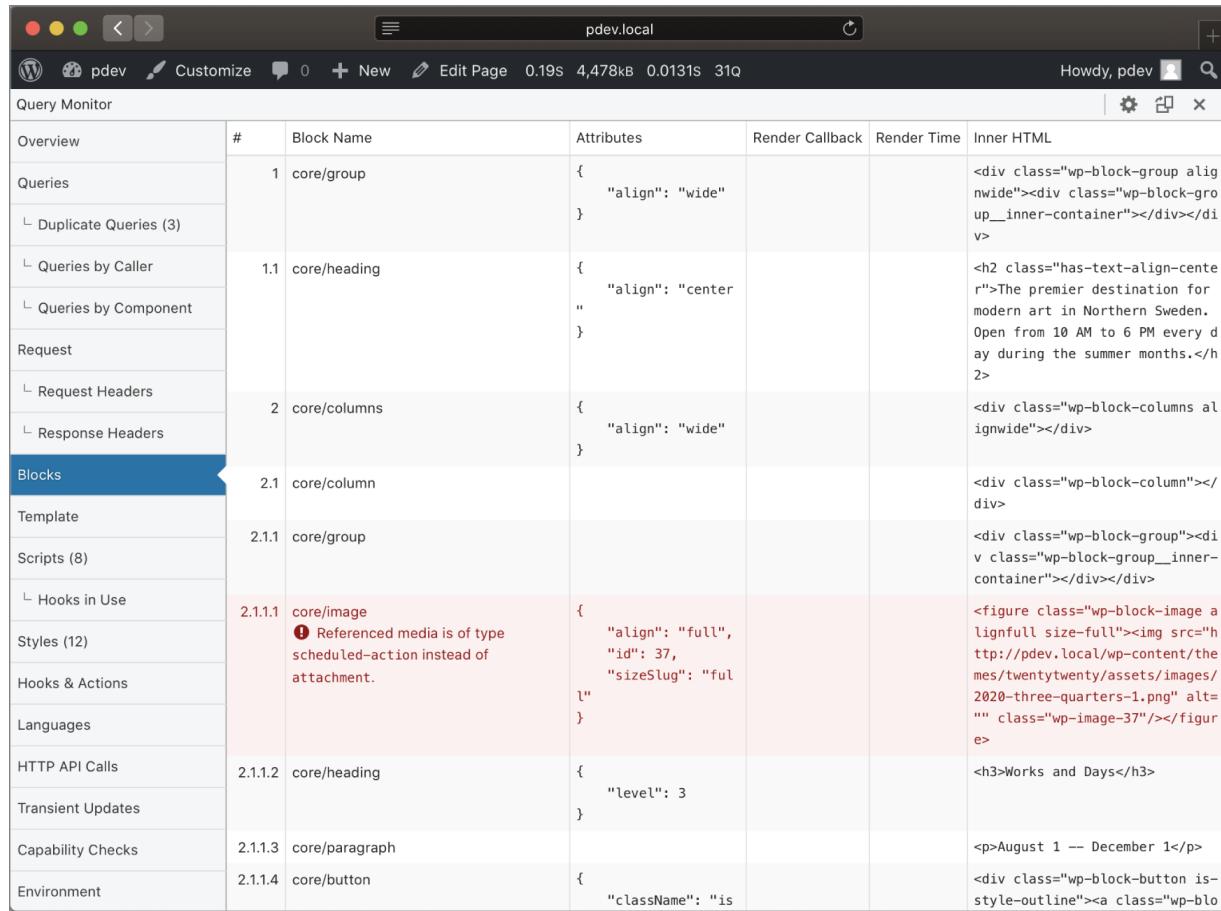
Overview		Query	Caller	Component	Rows	Ti
		All	All	All	All	Ti
Queries		1	SELECT option_name, option_value FROM wp_options WHERE autoload = 'yes'	wp_load_alloptions()	Core	213 0
└ Queries by Caller		2	SELECT option_value FROM wp_options WHERE option_name = 'WPLANG' LIMIT 1	get_option()	Core	0 0
└ Queries by Component		3	SELECT * FROM wp_users WHERE user_login = 'pdev' LIMIT 1	WP_User::get_data_by()	Core	1 0
Request		4	SELECT user_id, meta_key, meta_value FROM wp_usermeta WHERE user_id IN (1) ORDER BY umeta_id ASC	update_meta_cache()	Core	32 0
└ Request Headers		5	SELECT option_value FROM wp_options WHERE option_name = 'can_compress_scripts' LIMIT 1	get_option()	Core	1 0
└ Response Headers		6	SELECT option_value FROM wp_options WHERE option_name = 'theme_switched' LIMIT 1	get_option()	Core	0 0
Admin Screen		7	SELECT option_value FROM wp_options WHERE option_name = 'db_upgraded' LIMIT 1	get_option()	Core	0 0
Scripts (40)		8	SELECT option_value FROM wp_options WHERE option_name = 'site_transient_update'	get_option()	Core	1 0
└ Hooks in Use						
Styles (25)						
Hooks & Actions						
Languages						
HTTP API Calls						
Transient Updates						
Capability Checks						
Environment						
Conditionals						

FIGURE 15-2: Query Monitor interface

As shown in [Figure 15-3](#), you can see all of the database queries that ran on the current page, with multiple ways to filter and sort the results. This is super helpful when you want to narrow things down to only what your plugin is doing.

When Query Monitor is opened on a post or page that uses blocks, a new Blocks section becomes available and allows you to see every block as it is used. Here you can see that some of the default content for the Twenty Twenty theme is referencing the wrong media, and Query Monitor is able to identify that and alert you that something is incorrect.

Query Monitor is an essential part of every professional WordPress plugin developer's local development environment. You can install it via the WordPress admin's Add Plugin page and learn more at querymonitor.com.



The screenshot shows the WordPress Query Monitor interface. The left sidebar has a 'Blocks' tab selected. The main table lists database queries with columns for #, Block Name, Attributes, Render Callback, Render Time, and Inner HTML. The 'Inner HTML' column shows the rendered HTML for each query, including a heading, columns, and a figure element.

Overview	#	Block Name	Attributes	Render Callback	Render Time	Inner HTML
Queries	1	core/group	{ "align": "wide" }			<div class="wp-block-group alignwide"><div class="wp-block-group_inner-container"></div></div>
└ Duplicate Queries (3)	1.1	core/heading	{ "align": "center" }			<h2 class="has-text-align-center">The premier destination for modern art in Northern Sweden. Open from 10 AM to 6 PM every day during the summer months.</h2>
└ Queries by Caller	2	core/columns	{ "align": "wide" }			<div class="wp-block-columns alignwide"></div>
└ Queries by Component	2.1	core/column				<div class="wp-block-column"></div>
Request	2.1.1	core/group				<div class="wp-block-group"><div class="wp-block-group_inner-container"></div></div>
└ Request Headers	2.1.1.1	core/image	❗ Referenced media is of type scheduled-action instead of attachment.	{ "align": "full", "id": 37, "sizeSlug": "full" }		<figure class="wp-block-image alignfull size-full"></figure>
└ Hooks in Use	2.1.1.2	core/heading		{ "level": 3 }		<h3>Works and Days</h3>
└ Styles (12)	2.1.1.3	core/paragraph				<p>August 1 -- December 1</p>
└ Hooks & Actions	2.1.1.4	core/button		{ "className": "is-style-outline" }		<div class="wp-block-button is-style-outline"><a class="wp-blo
Languages						
HTTP API Calls						
Transient Updates						
Capability Checks						
Environment						

FIGURE 15-3: Database queries

SUMMARY

Entire books have been written about debugging code. As a general topic, this chapter is not written to cover every possible avenue but rather to present you with the basic tools available and to give you a look into the debugging features that WordPress specifically provides.

Some of the most important things you can do are stay informed about changes to WordPress, and constantly be debugging your plugins with the latest WordPress versions. Your plugins can use the most up-to-date functionality, be less prone to compatibility issues, and make users happy by taking advantage of the newest WordPress features. All these things require little time and will make your plugins great in the long term.

16

The Developer Toolbox

WHAT'S IN THIS CHAPTER?

- Using the WordPress core as a reference
- Understanding inline documentation
- Exploring popular core files and functions
- Using community resources and websites
- Learning about external tool websites
- Creating a developer toolbox

When developing plugins for WordPress, it's important to have a good list of resources to help guide you in the right direction. This list of resources is the developer's toolbox. In this chapter, you'll learn the most popular and helpful resources available for plugin development. You'll also review tools that every developer should use to help optimize the process of plugin development.

CORE AS REFERENCE

The best reference when developing plugins for WordPress is the core WordPress code. What better way to learn about the inner workings of WordPress, and discover new features, than to explore the code that powers every plugin you develop? Understanding how to navigate through the core WordPress code is a valuable resource when developing professional plugins. Also, contrary to online resources and the Codex, the core is always up-to-date.

PHP Inline Documentation

Many of the core WordPress files feature inline documentation. This documentation, in the form of a code comment, gives specific details on

how functions and code work. All PHP inline documentation is formatted using inspiration from the PHPDoc standard for PHP commenting, meaning there are some differences between the PHPDoc standard and the WordPress implementation of inline documentation. The following comment sample is the standard template for documenting a WordPress function:

```
/**  
 * Short Description  
 *  
 * Long Description  
 *  
 * @package WordPress  
 * @since version  
 *  
 * @param type $varname Description  
 * @return type Description  
 */
```

Inline documentation is an invaluable resource when exploring functions in WordPress. The comment includes a short and long description, detailing the purpose of a specific function. It also features the WordPress version in which it was added. This helps determine what new functions are added in each release of WordPress.

Parameters are also listed, along with the data type of the parameter and a description of what the parameter should be. The return type and description are also listed. This helps you understand what value a specific function will return. For example, when creating a new post in WordPress, the newly created post ID would be returned if successful.

Now look at the following real inline documentation for the `delete_option()` function:

```
/**  
 * Removes option by name. Prevents removal of protected  
 * WordPress options.  
 *  
 * @since 1.2.0  
 *  
 * @global wpdb $wpdb WordPress database abstraction object.  
 *  
 * @param string $option Name of option to remove. Expected  
 * to not be SQL-escaped.
```

```
 * @return bool True, if option is successfully deleted.  
 False on failure.  
 */  
 function delete_option( $option ) {
```

The inline documentation features a clear description of the purpose of this function. You can see that the function is part of the WordPress package and was added in version 1.2.0. The comment also lists any PHP globals that are used within the function or method, which can include an optional description of the global.

The only parameter required for this function is `$option`, which is described as the option name. Finally, the return value is boolean; `True` if successful and `False` on failure.

Another important tag that may exist within the inline documentation is the `@deprecated` tag. A deprecated function in WordPress is a function that should no longer be used. Typically, if a function is `@deprecated`, a new function will exist that replaces it. Let's look at an example of the `@deprecated` tag for the `get_the_author_description()` function.

```
/**  
 * Retrieve the description of the author of the current  
 post.  
 *  
 * @since 1.5.0  
 * @deprecated 2.8.0 Use get_the_author_meta()  
 * @see get_the_author_meta()  
 *  
 * @return string The author's description.  
 */  
 function get_the_author_description() {
```

As you can see, the `@deprecated` tag exists within the inline documentation. In this example, the appropriate new function to use is the `get_the_author_meta()` function. The deprecated function was added to WordPress in 1.5.0 and deprecated in the 2.8.0 release. This is especially important information to verify that your plugin is using the most up-to-date functions within the WordPress core software.

For more information on the PHP Documentation Standards, visit the WordPress Handbook section at

<https://make.wordpress.org/core/handbook/best-practices/inline-documentation-standards/php>.

JavaScript Inline Documentation

As with PHP, JavaScript files also include templated code documentation. WordPress follows the JSDoc 3 standard for inline JavaScript documentation. The format of the JSDoc 3 standard is similar to the WordPress PHP format, but there are some slight differences. To learn more about WordPress commenting in JavaScript, visit the handbook at <https://make.wordpress.org/core/handbook/best-practices/inline-documentation-standards/javascript>.

Inline documentation is an ongoing process. All new functions added to WordPress are documented using this process. Helping to document existing functions is a great way to dive into core contributing to WordPress.

Finding Functions

Now that you understand how to use inline documentation to learn functions in WordPress, you need to know how to find those functions. To start, make sure you have downloaded the latest version of WordPress to your local computer. You will be searching through these code files for functions.

Every core file, excluding images, can be viewed in a text editor program. When choosing a text editor to use, make sure it supports searching within files. You can find an extensive list of text editors on the Codex at <https://wordpress.org/support/article/glossary/#text-editor>.

When searching through the core WordPress files for a specific function, you need to make sure calls to that function are filtered out or you may get hundreds of results. The easiest way to do this is to include the word *function* at the start of your search. For example, to find `wp_insert_post()`, simply search for *function wp_insert_post*.

NOTE *Remember, not everything in WordPress is a function. If you don't get any results, remove the word function from your search. Also remember to search all files (*.*) and not just .txt files, which many text editors default to.*

Common Core Files

Many of the functions you use in your plugins are located in specific core files. Exploring these files is a great way to find new and exciting functions to use in your plugins.

The `wp-includes` folder features many of the files used for public-side functions: that is, functions used on the public side of your website.

formatting.php

The `formatting.php` file contains all WordPress API formatting functions, such as the following:

- `esc_*`(): Includes all escaping functions in this file
- `is_email`(): Verifies that an email address is valid
- `wp_strip_all_tags`(): Strips all HTML tags, including script and style, from a string
- `get_url_in_content`(): Extracts and returns the first URL found in a text string

functions.php

The `functions.php` file contains the main WordPress API functions. Plugins, themes, and the WordPress Core use these functions, for example:

- `*_option`(): Adds, updates, deletes, and retrieves options
- `current_time`(): Retrieves the current time based on the time zone set in WordPress
- `wp_nonce:*`(): Creates nonce values for forms and URLs

- `wp_upload_dir()`: Retrieves array containing the current upload directory's path and URL
- `is_wp_version_compatible()`: Checks compatibility with the current version of WordPress
- `is_php_version_compatible()`: Checks compatibility with the current version of PHP

pluggable.php

The `pluggable.php` file contains core functions that you can redefine in a plugin. This file is full of useful functions for your plugins, for example:

- `get_userdata()`: Retrieves all user data from the specified user ID
- `wp_get_current_user()`: Retrieves user data for the currently logged-in user
- `get_avatar()`: Retrieves a user's avatar
- `wp_mail`: Is the main function for sending email in WordPress
- `wp_redirect()`: Redirects to another page
- `wp_rand()`: Generates a random number

plugin.php

The `plugin.php` file contains the WordPress Plugin API functions, such as the following:

- `add_action()`: Executes this hook at a defined point in the execution of WordPress
- `add_filter()`: Uses this hook to filter prior to saving in the database or displaying on the screen
- `plugin_dir_*`(): Functions to determine a plugin's path and URL
- `register_activation_hook()`: Is called when a plugin is activated
- `register_deactivation_hook()`: Is called when a plugin is deactivated

- `register_uninstall_hook()`: Is called when a plugin is uninstalled and `uninstall.php` does not exist in the plugin directory

post.php

The `post.php` file contains the functions used for posts in WordPress, as follows:

- `wp_*_post()`: Functions for creating, updating, and deleting posts
- `get_posts()`: Returns a list of posts based on parameters specified
- `get_pages()`: Returns a list of pages based on parameters specified
- `*_post_meta()`: Functions to create, update, delete, and retrieve post metadata
- `register_post_type()`: Registers custom post types in WordPress
- `get_post_types()`: Retrieves a list of all registered post types
- `set_post_thumbnail()`: Sets the post thumbnail (featured image) for a specific post
- `wp_delete_attachment()`: Trashes or deletes an attachment based on the post ID

user.php

The `user.php` file contains the functions used for users in WordPress, as follows:

- `get_current_user_id()`: Returns the ID of the currently logged-in user
- `*_user_meta()`: Functions for adding, updating, deleting, and retrieving user metadata
- `wp_*_user()`: Functions for adding and updating a WordPress user

This is just a small list of the more popular functions and their locations. Many more functions are available to use when developing your plugins. When a new version of WordPress is released, it's always fun to explore

these files to see what functions have been added and are available for use in your plugins.

PLUGIN DEVELOPER HANDBOOK

One of the most important online resources for plugin development is the WordPress Plugin Developer Handbook. The handbook is the official online resource for plugin development documentation and is located at <https://developer.wordpress.org/plugins>.

The handbook is a project created and supported by the WordPress Docs Team. This team is a volunteer group of contributors that help maintain the information contained within the handbook. It's a living resource in that it's continually updated to be as accurate as possible.

This resource is packed full of information about using, and developing with, WordPress. It includes an extensive feature and function reference with some helpful tutorials and examples demonstrating the more common functions in WordPress.

Navigating the Handbook

The Plugin Developer Handbook covers a wide range of plugin development topics. To quickly find a topic of interest, you can use the search field. There is also extensive navigation on the left with topics and subtopics covering a large variety of plugin development material.

The handbook is an exceptional resource when developing WordPress plugins. If you are interested in contributing to the handbook, reach out to the WordPress Docs Team at <https://make.wordpress.org/docs>.

Code Reference

Another invaluable tool on wordpress.org for plugin developers is the Code Reference. The Code Reference is a detailed handbook of WordPress functions, classes, methods, and hooks. You can easily search using the search field and filter the results by the type of item you are looking for.

The Code Reference is largely generated using the inline comments that we reviewed earlier in this chapter. Let's look at the `add_option()` function as

an example:

https://developer.wordpress.org/reference/functions/add_option.

As shown in [Figure 16-1](#), the top of the code reference page shows the function with all the available parameters. Next is a detailed description of exactly what the function does, in this case creating a new option value in the database. The parameters are then listed with each parameter type, whether it is required or not, a short description explaining the parameter, and the default value.

```
add_option( string $option, mixed $value = '',
            string $deprecated = '',
            string/bool $autoload = 'yes' )
```

[FIGURE 16-1](#): Function with parameters

Next the return value is shown, if one exists. The source file is identified, so you know where the function is defined, in this case `wp-includes/option.php` as well as the code defining the function. Next we have the changelog, which identifies what version of WordPress introduced this function. The related section shows “Uses” and “Used by” functions. The final section is the “User Contributed Notes” section. This is the one area where a user can log in and submit example code and notes to expand the entry. As you can see, this visual resource may be an easier way to digest the information that exists within the WordPress Core software.

Visit the Code Reference at

<https://developer.wordpress.org/reference.>

CODEX

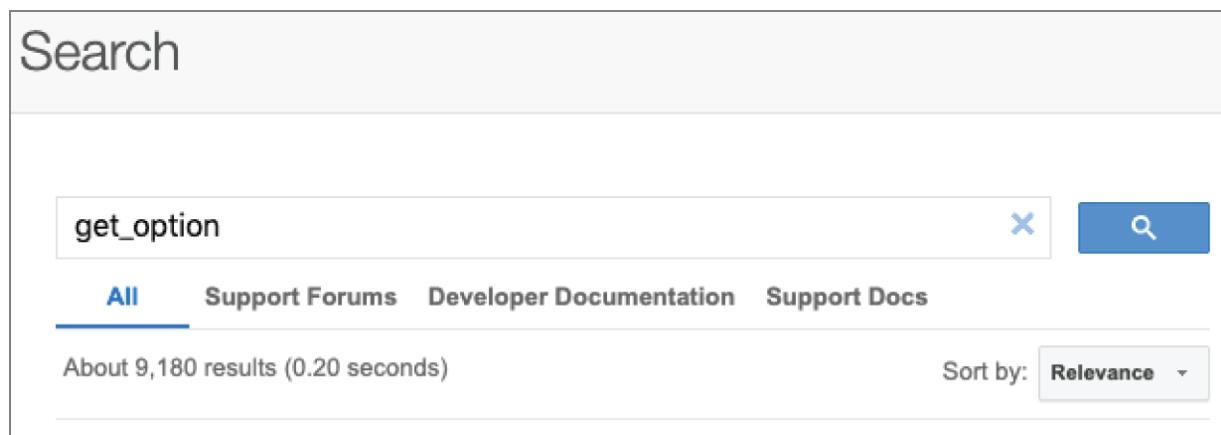
Another important resource, yet slightly older, is the WordPress Codex. The Codex is an online wiki for WordPress documentation and is located at <https://codex.wordpress.org.>

The Codex is packed full of information about using, and developing with, WordPress. It includes an extensive function reference with some helpful

tutorials and examples demonstrating the more common functions in WordPress.

Searching the Codex

You can search the Codex in a few different ways. The most common way is to use the Codex search located at <https://wordpress.org/search> or to enter your search terms in the search box located in the header of [WordPress.org](https://wordpress.org). The default search for the Codex is everything, but you can also search the Support Forums, Developer Documentation, or Support Docs, as shown in [Figure 16-2](#).



[FIGURE 16-2:](#) Codex search options

The Codex also features an extensive glossary. This can help you become familiar with terms used in the Codex, as well as in WordPress. You can access the glossary at

<https://wordpress.org/support/article/glossary>.

The Codex home page also features an index of articles organized by topic. The articles are ordered by level of difficulty, with an article specific to the latest version of WordPress near the top. This article details new features, functions, changes, and so on, in the latest version of WordPress. It's always helpful to read this article to become familiarized with any changes to functions and methods for developing plugins in WordPress.

Function Reference

One big benefit of the Codex is the function reference section located at https://codex.wordpress.org/Function_Reference. This section lists

all functions in the Codex for the more popular WordPress API functions. This page is a must-bookmark for any WordPress plugin developer.

Each individual function reference page contains a description of the function, basic usage example, parameters required for the function, and return values. Think of these function reference pages as quick and easily readable inline documentation on a function. Most function pages also feature an example, or more, demonstrating practical uses of that particular function.

NOTE *The Codex is a great resource, but by no means is guaranteed to be accurate or up-to-date. Remember, the WordPress Core is the only resource that is always 100 percent up-to-date.*

TOOL WEBSITES

Many different websites are available to help in researching and understanding specific functionality in WordPress. These sites can help as new versions of WordPress are released with new functionality that can be used in your plugins.

PHPXref

PHPXref is a cross-referencing documentation generator. Quite simply, it is a developer tool that can scan a project directory and translate the files processed into readable cross-referenced HTML files. It automatically processes PHPDoc commenting to produce documentation for the functions included.

An online hosted version of PHPXref is also available, which is more specifically for WordPress. The online version is located at <https://phpxref.ftwr.co.uk/wordpress>.

Visiting the WordPress PHPXref site, you'll be confronted with what looks like a Windows Explorer layout, as shown in [Figure 16-3](#).



FIGURE 16-3: WordPress PHPXref

As you can see, the standard WordPress folder displays on the left with the core subdirectories and root files listed. As an example, click into the `wp-includes` directory and click the `plugin.php` file. Clicking this link brings up a summary view of the current file selected, in this case `plugin.php`. This summary view has useful information including a list of every function in the file, as shown in [Figure 16-4](#).

[[Index](#)]

[[Hide Explorer](#)]

/wp-includes/ -> plugin.php

[[Source view](#)] [[Print](#)] [[Project Stats](#)]

The plugin API is located in this file, will then be run when the action or f methods, you'll need to pass an array

File Size: 931 lines (32 kb)

Included or required: [1 time](#)

Referenced: 0 times

Includes or requires: 0 files

Defines 28 functions

- [add_filter\(\)](#)
- [has_filter\(\)](#)
- [apply_filters\(\)](#)
- [apply_filters_ref_array\(\)](#)
- [remove_filter\(\)](#)
- [remove_all_filters\(\)](#)
- [current_filter\(\)](#)
- [current_action\(\)](#)
- [doing_filter\(\)](#)
- [doing_action\(\)](#)
- [add_action\(\)](#)
- [do_action\(\)](#)
- [did_action\(\)](#)
- [do_action_ref_array\(\)](#)
- [has_action\(\)](#)
- [remove_action\(\)](#)
- [remove_all_actions\(\)](#)
- [apply_filters_deprecated\(\)](#)
- [do_action_deprecated\(\)](#)
- [plugin_basename\(\)](#)
- [wp_register_plugin_realpath\(\)](#)
- [plugin_dir_path\(\)](#)
- [plugin_dir_url\(\)](#)
- [register_activation_hook\(\)](#)
- [register_deactivation_hook\(\)](#)
- [register_uninstall_hook\(\)](#)
- [_wp_call_all_hook\(\)](#)
- [_wp_filter_build_unique_id\(\)](#)

FIGURE 16-4: Function list

Getting a top-level glance at all functions in a WordPress core file is a great way to find new functions and even locate existing functions for reference. Clicking any listed function takes you to the section of the page detailing the usage of the function. This information is extracted from the inline documentation saved in the WordPress Core file. If the function is not documented in WordPress, this section will be empty. You can also easily view the source of any file by clicking the Source View link near the header of the page.

It's easy to see how useful the WordPress PHPXref site is for a developer. This is another required tool to add to your resource arsenal.

Hooks Database

The WordPress hooks database, which was created and is supported by Adam Brown, is the essential resource for discovering hooks in WordPress. Adam built a system that digs through all WordPress core files and extracts every action and filter hook that exists in WordPress. He has been indexing these values since WordPress 1.2.1 and updates the hooks database with each new major version of WordPress.

One of the best features of the hooks database is you can view all new hooks added in each version of WordPress. Hooks are one of the most powerful features that you can use when creating plugins in WordPress. Clicking any hook name produces a hook detail screen showing where the hook is defined in the WordPress Core code.

To visit the hooks database, visit https://adambrown.info/p/wp_hooks.

COMMUNITY RESOURCES

There are also many different community resources available to help with WordPress development. These resources can help you expand your knowledge of plugin development, troubleshoot plugin issues, and work with new features in WordPress.

Make WordPress

The Make WordPress section of WordPress.org is a collection of websites and team groups dedicated to contributing to different parts of the WordPress project. You can view all teams on the official website at <https://make.wordpress.org>. These are important WordPress teams for plugin developers to follow:

- **Core:** This is the core team that creates the WordPress software. If you are interested in contributing to the core development of WordPress, or just want to stay in the loop on active discussions, this is the team for you.
- **Design:** The team is dedicated to designing and developing the user interface of WordPress. This is the perfect team for designers and user experience (UX)-focused users.
- **Accessibility:** Accessibility is an extremely important part of all website development. This team focuses on making WordPress accessible to everyone on the Internet.
- **Polyglots:** WordPress is used by millions of websites around the globe. This team focuses on translating WordPress into as many languages as possible.
- **Plugins:** The plugin review team is tasked as the gatekeepers for all plugins released in the official WordPress Plugin Directory. This team is especially important to keep up-to-date on development, security, and guidelines for building and releasing plugins.
- **Documentation:** A critical component to all software projects is good documentation. This is the WordPress team dedicated to that effort.

As you can see, there are many different teams dedicated to the growth and success of the WordPress project. The teams consist largely of volunteers who help coordinate and run project initiatives. You can view an aggregated list of team meetings at <https://make.wordpress.org/meetings>.

Support Forums

WordPress.org features a large support forum for topics ranging from using WordPress to plugin development. You can visit the support forums at <https://wordpress.org/support>.

Multiple forum sections can help expand your knowledge on plugin development and can support any public plugins you have released. The following are the forum sections specific to plugins:

- <https://wordpress.org/support/forum/wp-advanced>: Discussions are more advanced and complex than usual.
- <https://wordpress.org/support/forum/multisite>: Anything and everything regarding the Multisite feature.
- <https://wordpress.org/support/forum/plugins-and-hacks>: Plugin support questions. If you release a plugin to the Plugin Directory, users can submit support issues specific to your plugin.

WordPress Slack

Often it's nice to have a live conversation when seeking help for plugins or about WordPress development. WordPress uses Slack for live chat and has a number of active chat rooms or channels. To join the WordPress Slack Workspace, you need to follow the instructions at <https://make.wordpress.org/chat>.

- **#core**: This is the main channel for WordPress Core discussions. Room conversations focus primarily on the development of the WordPress software and the management of the WordPress.org systems. This room is not for general WordPress discussions, but rather specific to core development or bugs and issues found within the core. Most of the core WordPress developers and contributors are in this room and willing to help you.
- **#pluginreview**: This channel is dedicated to the discussion of technical reviews on plugins submitted to the WordPress Plugin Directory on WordPress.org. This is a great place to chat with the Plugin Review Team regarding publishing your plugin on WordPress.org.

These Slack channels are an awesome resource for real-time help when developing plugins in WordPress. Many WordPress experts hang out in these rooms and enjoy helping others learn WordPress.

WordPress Development Updates

When developing plugins, and releasing to the public, you need to stay current on new features and functionality coming in the new version of WordPress. This can help you verify not only if your plugin is compatible with the latest version but also what new features your plugins can take advantage of. One of the best ways to track WordPress development is through the Make WordPress Core site at

<https://make.wordpress.org/core>.

The Make WordPress Core site is a blog that focuses on the core development of WordPress. You can also find out about the weekly developer chats that take place in the Slack channel #core. These chats discuss the current status of the upcoming WordPress version, feature debates and discussions, and much more.

WordPress Ideas

WordPress.org features an ideas section for creating and rating ideas for future versions of WordPress. This is actually a great resource for gathering ideas for plugins. Many of the ideas submitted could be created using a plugin. The more popular an idea is, the more popular the plugin will most likely be.

The WordPress Ideas area is at <https://wordpress.org/ideas>.

Community News Sites

There are some great websites that focus on WordPress developer-related news and articles. Many of these sites feature development-focused tutorials and tips that can be used when creating your plugins. The following sections explore useful community news sites that are available.

WordPress News

The [WordPress.org](https://wordpress.org) Blog, called WordPress News, is the best place to stay current on WordPress news. The blog centers around WordPress core release information but also includes other important WordPress-related news. You can view the official blog at <https://wordpress.org/news/> and subscribe to the feed at <https://wordpress.org/news/feed>.

WordPress Planet

WordPress Planet is a blog post aggregator located on WordPress.org. This includes posts from WordPress core contributors and active community members. This news feed, together with the WordPress.org Blog feed, is also featured in the WordPress Events and News Dashboard widget of every WordPress-powered website by default.

WordPress Planet is located at <http://planet.wordpress.org>.

Post Status

As a premium club, Post Status is a news website dedicated to the WordPress ecosystem for professionals and enthusiasts. Post Status members are invited to a member-only Slack Workplace that is an amazing resource when developing professional plugins for WordPress. There are multiple Slack channels available for help including #development, #javascript, and #club. These channels feature many advanced WordPress plugin developers available to help answer questions or give general guidance.

You can learn more about the Post Status Club by visiting its website at <https://poststatus.com>.

Know the Code

Know the Code provides online training courses for anyone who wants to specialize in WordPress development. With hands-on code building labs, the goal of Know the Code is to empower every WordPress developer to learn deeply about architecture, programming fundamentals, testing, validation, code quality, and more.

You can visit Know the Code at <https://knowthecode.io>.

LinkedIn Learning

LinkedIn Learning features some amazing online courses taught by professionals in the industry. Topics range from basic plugin development to advanced courses including actions and filters, creating editor blocks, custom workflows, building apps with the REST API, and more. Visit

<https://www.linkedin.com/learning/topics/wordpress> to find out more.

Twitter

Twitter is also a great resource for following WordPress developers, contributors, and the entire community. More specifically, the Twitter account @wordpress is the official WordPress account. This is a quick and easy way to stay current with news and information about WordPress. You can also follow your favorite book authors on Twitter at @williamsba, @justintadlock, and @jjj!

Local Events

Another great resource is local WordPress events. When learning to create plugins in WordPress, it can help to find other enthusiastic developers near you to learn from and work with.

WordCamps are locally organized conferences covering anything and everything WordPress. Many WordCamps feature plugin-development-specific presentations given by some of the top plugin developers in the community. These presentations are a great way to learn new and advanced techniques to use in your plugins. To find a WordCamp near you, visit <https://central.wordcamp.org>.

WordPress Meetups are also a great way to meet local developers in your area. Meetups generally happen more often, typically on a monthly basis, and are a local gathering of WordPress enthusiasts. Meetups are also generally smaller, more focused groups allowing for more in-depth and personal conversations to take place. To find a local WordPress Meetup, visit <https://www.meetup.com/pro/wordpress>.

TOOLS

When developing plugins for WordPress, you want to use specific tools to make your life much easier.

Browser

WordPress is web software; therefore, you will spend much of your time debugging plugins in your web browser. Some browsers stand above the rest when it comes to developer features. The two more popular development browsers are Firefox and Google Chrome. Both of these browsers feature development tools and can be expanded with additional tools to make debugging and troubleshooting much easier.

Firefox features powerful web development tools called the Firefox Developer Tools. These tools enable you to easily edit, debug, and monitor HTML, CSS, and even JavaScript. Firefox also supports Extensions enabling you to enhance the features in Firefox. You can learn more about Firefox Developer Tools at <https://developer.mozilla.org/en-US/docs/Tools>.

Google Chrome is also a great development browser. Chrome features a built-in set of tools called Chrome DevTools. These tools can enable you to edit and debug HTML, CSS, and JavaScript in real time. You can also install extensions in Chrome that add additional functionality to the browser. You can learn more about Chrome DevTools at <https://developers.google.com/web/tools/chrome-devtools>.

Editor

Creating plugins for WordPress is as simple as creating a PHP file. PHP files are actually text files with a .php extension. Because of this, you can develop PHP files using any basic text editor. Although text editors can certainly get the job done, they won't offer the more advanced features such as syntax highlighting, function lookup, spell-check, and so on.

NetBeans IDE

NetBeans IDE is a popular editor that is an open source development environment that runs on Java. Because of this, it can run on any platform that supports the Java Virtual Machine including Windows, macOS, Linux, and Solaris. NetBeans supports PHP with syntax highlighting, PHP debugging using Xdebug, remote and local project development, and many more features. For more information and to download NetBeans, visit <https://netbeans.org>.

PhpStorm

PhpStorm is another popular IDE that supports Mac, Windows, and Linux. A major reason this IDE is great is the direct WordPress integration support. PhpStorm also has built-in developer tools including version control systems, support for remote deployments, database/SQL, command-line tools, Docker, and Composer support. To learn more and download PhpStorm, visit <https://www.jetbrains.com/phpstorm>.

Notepad++

Notepad++ is a popular open source text editor that runs on Windows, macOS, and Linux. The editor is a lightweight text editor, similar to standard Notepad, but offers many features including syntax highlighting, macros and plugins, autocompletion, regular expression find and replace, and more. To download Notepad++, visit <http://notepad-plus-plus.org>.

TextMate

TextMate is a GUI text editor for macOS. It's a popular editor among developers because it features some programming-centric features. Some of these features include nested scopes, bundles (snippet, macro, and command groupings), project management, and more. You can download TextMate at <https://macromates.com>.

Sublime Text

Another popular text editor is Sublime Text. One popular feature is the Goto Definition, which allows you to quickly find where a class, method, or function is defined within the WordPress code in a single click. This is extremely handy when building plugins as you can quickly consult the core code as needed. Sublime Text is available for Mac, Windows, and Linux at <https://www.sublimetext.com>.

Visual Studio Code

Developed by Microsoft, Visual Studio Code is another great popular, free, and open source option. Visual Studio Code is lightweight like Sublime Text but offers many of the same features as bigger IDEs like PhpStorm. The editor supports Windows, Mac, and Linux at <https://code.visualstudio.com>.

Deploying Files with FTP, SFTP, and SSH

When developing plugins, you need to decide whether you plan to run a local instance of WordPress on your computer or use an external web server. If using an external server, you need to use some method to push your files to your server. The most popular method is using SFTP.

File Transfer Protocol (FTP) is a standard network protocol used to copy files from your computer to another computer, or a web server in this case. For FTP, FileZilla is a free, open source FTP client that works on Windows, macOS, and Linux. You can learn more about FileZilla and can download the client at <https://filezilla-project.org>.

Secure File Transfer Protocol (SFTP) is also a popular method for deploying files to a server. The major difference between FTP and SFTP is that SFTP is encrypted. This means any account info, usernames, passwords, and files you transfer are sent over an encrypted transport. Many popular FTP clients, including FileZilla, also support SFTP.

Secure Shell (SSH) is a third option for transferring files to a web server. SSH is more than just a way to transfer files. For example, you can interact with MySQL using SSH. A popular SSH client is PuTTY, which runs on Windows and UNIX platforms and can be found at <https://www.putty.org>. Mac users can use the built-in shell called Terminal when working with SSH.

WARNING *Standard FTP is no longer a recommended approach as your credentials are passed in plain text, meaning they are not encrypted and can easily be compromised.*

phpMyAdmin

On occasion you may need to directly work with the WordPress database. The most common method to do this is by using phpMyAdmin, which is a free tool, written in PHP, which provides a simple web interface to view and manage a MySQL database. Most web hosts have this tool installed by default.

Using phpMyAdmin, you can easily view what data your plugin adds to WordPress tables. This can significantly help speed up the debug process. You can also run phpMyAdmin on your computer by downloading the software package at <https://www.phpmyadmin.net/downloads>.

NOTE *MySQL can also be administered through SSH; however, SSH is a command-line interface and a bit more advanced than using the web interface of phpMyAdmin.*

SUMMARY

As a plugin developer, you need a solid set of tools in your developer toolbox. Every developer's toolbox is unique and customized to your tastes and server setup. Becoming comfortable with your developer tools can help you be comfortable when creating professional plugins in WordPress. If you don't have a set of preferred tools, you should ask around and see what other developers use. It's always fun to talk with other programmers to see what tools they use and recommend.

When developing, news and community websites are just as important as your tools. These sites can help you expand your knowledge on plugin development, learn about new features and proper usage of those features, and become a part of the greatest community on the Web: the WordPress Community. Code on!

INDEX

A

<a href> link, [67](#)
abs() function, [82](#)
absint() function, [332](#)
\$abs_rel_path parameter, [262](#)
\$accepted_args parameter, [107](#), [118](#)
access, limiting, [225](#)–228
Accessibility team, [424](#)
action hooks, [106](#)–117
actions, [107](#)–108
activate() method, [22](#)
Active status, for plugins, [10](#)
add_action() function, [107](#)–108, [109](#), [126](#), [127](#), [419](#)
add_blog_option() function, [342](#)
add_cap() function, [230](#)–231
add_filter() function, [118](#), [119](#), [120](#), [122](#), [126](#), [127](#), [419](#)
add_menu_page() function, [32](#)
add_meta_box() function, [188](#)–189
add_network_option() function, [342](#)
add_option() function, [36](#)–37, [41](#), [42](#)
add_options_page() function, [34](#)–35
add_permastruct() function, [392](#)
add_post_meta() function, [184](#)–185, [186](#)

add_rewrite_rule() function, [389](#)
add_rewrite_tag() function, [392](#)
add_role() function, [228](#)–230
add_settings_error() function, [51](#)
add_settings_field() function, [44](#)–45, [52](#)
add_settings_section() function, [44](#)–45
add_shortcode() function, [325](#), [327](#), [359](#), [367](#)
add_site_meta() function, [343](#)
add_submenu_page() function, [33](#)
add_user_meta() function, [218](#)–219
add_user_to_blog() function, [344](#), [345](#)
admin bar, [73](#)
admin user, [206](#), [227](#)–228
Administrator role, [224](#)
admin_menu action hook, [115](#)
admin_url() function, [21](#)
\$after parameter, [196](#)
Ajax, [145](#)–146
all-in-one methods, [91](#)–93
Angular, [146](#)
anonymous functions, using hooks with, [127](#)–128
Apache, [385](#), [403](#)
application passwords, [286](#)
apply_filters() function, [117](#)–118, [119](#), [121](#), [128](#)
apply_filters_ref_array() function, [121](#), [128](#)
arbitrary text strings, [82](#)–83

Archived status (Multisite), [316](#)
\$args parameter, [106](#), [117](#), [183](#), [193](#), [226](#), [236](#), [240](#), [287](#), [295](#)
array_map() function, [89](#)
arrays, [37](#), [39](#)–40, [89](#)
attachments, [176](#), [182](#)–183
\$attr parameter, [362](#)
attributes, escaping, [85](#)–86
auth_callback argument, [184](#)
authentication, [284](#)–286
Author role, [224](#)
author_can() function, [226](#)
authority, intention vs., [74](#)–75
autoload parameter, [40](#)–42
automatic mapping, [181](#)

B

Babel, [163](#)
Backbone, [146](#)
balanced tags, forcing, [86](#)
Basic Authentication handler plugin for WordPress, [284](#)–285
bbPress, [402](#)
\$before parameter, [196](#)
best practices, for plugins, [14](#)–17
block category, [154](#)
Block Directory, [173](#)–174
Block Library, [154](#)
block renderer, as REST API endpoint, [283](#)

block revisions, as REST API endpoint, [283](#)

blocks

about, [150](#)

defined, [154](#)

as REST API endpoint, [283](#)

WP-CLI scaffolding, [169](#)

blocks tag, [5](#)

`block_version()` function, [401](#)

Blog Pester plugin, [247](#)–250

body parameter, [296](#), [302](#)

`<body>` tag, [107](#)–108

`body_class` hook, [119](#)

Book Collection plugin, [177](#)–179

`boot()` method, [114](#), [127](#), [243](#)

brace style, [29](#)

broken plugins, [7](#)

browsers, [427](#)–428

BuddyPress, [4](#), [402](#)

built-in auto installer, [8](#)

bundled scripts, [139](#)–141

buttons, [66](#)–67

C

Cache API, [97](#). *See also* [caching](#)

caching, [98](#)–101

`$callback` parameter, [188](#), [359](#), [378](#)

`$callback_args` parameter, [188](#), [378](#)

capabilities. *See* [roles and capabilities](#)

`$capability(ies)` parameter, [226](#), [228](#)

categories, as REST API endpoint, [282](#)

Category taxonomy, [192](#)

chaining jQuery, [143](#)

`checked()` function, [54](#)

`$class` parameter, [119](#)

classes, [27](#)–28, [126](#)–127. *See also* specific classes

`$classes` parameter, [119](#)

`class_exists()` function, [401](#)

Classic Editor, [150](#), [151](#)

clients, [283](#)–289, [404](#)–405

client/server, [280](#), [289](#)

code, [130](#), [144](#)–145

code on demand constraint, [280](#)

Code Reference, [420](#)–421

Codex, [421](#)–422

coding standards

- about, [25](#)–26
- brace style, [29](#)
- documenting code, [26](#)–27
- double quotes, [28](#)
- indentation, [28](#)–29
- naming classes/methods, [27](#)–28
- naming files, [28](#)
- naming functions/variables, [27](#)
- shorthand PHP, [30](#)
- single quotes, [28](#)
- space usage, [29](#)–30
- SQL statements, [30](#)
- columns, selecting, [94](#)–95
- command line, [166](#), [278](#)
- comments, as REST API endpoint, [282](#)
- community, [7](#), [424](#)–427
- companion functions, [302](#)–303
- compatibility, debugging and, [399](#)–405
- Composer, [28](#)
- conditional tags, [197](#)–199

consistency

about, [63](#)

buttons, [66](#)–67

Dashicons, [64](#)–65

in folders, [16](#)

form fields, [67](#)–68

headings, [64](#)

messages, [65](#)–66

pagination, [69](#)–70

tables, [68](#)–69

using WordPress UI, [64](#)

content

about, [175](#)–176

building shortcodes with, [364](#)–366

creating custom post types, [176](#)–183

creating custom taxonomies, [191](#)–195

meta boxes, [187](#)–191

post metadata, [183](#)–187

post metadata plugin, [199](#)–204

post type plugin, [199](#)–204

taxonomy plugin, [199](#)–204

using custom taxonomies, [195](#)–199

content area, [155](#)

`content_url()` function, [21](#)

`$context` parameter, [188](#), [265](#)

Contributor role, [224](#)

\$control_callback parameter, [378](#)
cookies parameter, [296](#), [302](#)
#core, [425](#)
Core Developer Handbook, [3](#)
core files
 common, [418](#)–420
 modifying, [5](#)
Core team, [424](#)
count_many_users_posts() function, [216](#)
count_user_posts() function, [215](#)–216
count_users() function, [207](#)–208
create-guten-block toolkit, [170](#)–173
creating, reading, updating, and deleting (CRUD), [280](#)
cron
 about, [235](#)–236
 Blog Pester plugin, [247](#)–250
 Delete Comments plugin, [253](#)–258
 deleting post revisions weekly, [250](#)–252
 executing, [236](#)
 scheduling events, [236](#)–246
 true, [247](#)
crontab command, [247](#)
cross-site forgery request, [75](#)
CRUD (creating, reading, updating, and deleting), [280](#)
cryptographic nonces. *See* [nonces](#)
/css file, [16](#)

ctype_digit() function, [82](#)
CURL extension, [294](#)
current_action() function, [112](#)–113
current_filter() function, [122](#)–123
current_time() function, [418](#)
current_user_can() function, [73](#), [74](#), [225](#)–227
current_user_can_for_blog() function, [226](#)–227
Custom Post Type UI, [4](#)
custom roles, [224](#)
custom scripts
 about, [141](#)–142
 Ajax, [145](#)–146
 jQuery, [142](#)–143
 launching code on document ready, [144](#)–145
Customizer, [150](#)
customizing roles, [228](#)–233

D

dashboard, [31](#)–36
dashboard widgets
 about, [377](#)–378
 creating, [378](#)–380
 creating with options, [380](#)–384
Dashboard Widgets API, [2](#)

data

arrays of, [89](#)

cached, deleting, [99](#)

cached, loading, [99](#)

cached, saving, [98](#)–99

caching within plugins, [100](#)–101

receiving, [395](#)–396

responding to, [395](#)–396

sending, [395](#)

`$data` parameter, [92](#), [93](#), [98](#)

data sanitization

caching, [98](#)–101

formatting SQL statements, [90](#)–97

functions for, [81](#)–90

identifying tainted data, [80](#)

input data, [81](#)

need for, [78](#)–80

performance and, [97](#)

security and, [97](#)

data validation

- caching, [98–101](#)
- formatting SQL statements, [90–97](#)
- functions for, [81–90](#)
- identifying tainted data, [80](#)
- input data, [81](#)
- need for, [78–80](#)
- performance and, [97](#)
- security and, [97](#)

Database API, [3](#)

databases

- about, [350](#)
- Multisite-specific tables, [350–351](#)
- queries, [89–90](#)
- site-specific tables, [351](#)
- URLs in, [85](#)

`deactivate()` method, [23](#)

debugging

- about, [399, 405](#)
- compatibility and, [399–405](#)
- enabling, [406](#)
- error logging, [410–411](#)
- output, [406–410](#)

Query Monitor, [411–414](#)

default roles, [224](#)

Delete Comments plugin, [253–258](#)

`DELETE` method, [280](#)
`delete_blog_option()` function, [342](#)
Deleted status (Multisite), [316](#)
`delete_network_option()` function, [342](#)
`delete_option()` function, [36](#), [63](#)
`delete_post_meta()` function, [186](#)–187
`delete_site_meta()` function, [343](#)
`delete_transient()` function, [102](#)
deleting
 cached data, [99](#)
 expiring options, [102](#)
 options, [40](#)
 post metadata, [186](#)–187
 post revisions weekly, [250](#)–252
 posts using REST API, [309](#)–312
 roles, [230](#)
 user metadata, [220](#)
 users, [208](#)–209, [212](#)–213
deploying files with FTP, SFTP, and SSH, [429](#)
deprecation, [404](#)
description argument, [183](#), [209](#)
Design team, [424](#)
determining paths, [19](#)–21
developer toolbox. *See* [tools](#)
`did_action()` function, [112](#)
directories, for plugins, [9](#)–10

displaying-num class, [70](#)
display_name argument, [209](#), [213](#)
\$display_name parameter, [228](#)
<div> tag, [66](#)
do_action() function, [106](#), [108](#), [110](#), [128](#)
do_action_ref_array() function, [110](#)–111, [128](#)
document ready, launching code on, [144](#)–145
Documentation team, [424](#)
documenting code, [26](#)–27
dollar sign (\$), [144](#)
\$domain parameter, [262](#), [263](#), [265](#), [267](#), [336](#)
do_settings_fields() function, [59](#)
do_settings_sections() function, [47](#), [59](#)
do_shortcode() function, [367](#)–368
double quotes, [28](#)
downloading plugins, [4](#)
Drop-ins status, for plugins, [10](#)–11

E

_e() function, [263](#)
Easy Digital Downloads, [402](#)
echoing strings, [262](#)–271
ECMAScript, [161](#)
editing plugins, [9](#)
Editor role, [224](#)
editors, [428](#)–429
 tag, [86](#)–87

email strings, [84](#)

enabling

debugging, [406](#)

error logging, [410](#)

Multisite in WordPress, [317](#)–318

endpoints, [282](#)–288

enqueueing scripts, [135](#)–136

environment variables, [88](#)

equal comparison operator (==), [111](#)

error logging

about, [410](#)

enabling, [410](#)

log file overview, [411](#)

setting log file location, [411](#)

`error_log()` function, [306](#)

errors, with plugins, [7](#)

ES6, [163](#)–164

`esc_*`() function, [418](#)

escaping HTML/attributes, [85](#)–86

`esc_attr()` function, [46](#), [86](#), [88](#), [89](#)–90, [263](#), [274](#), [331](#)

`esc_attr_e()` function, [264](#)

`esc_attr_x()` function, [266](#)

`esc_html_()` function, [264](#)

`esc_html()` function, [85](#)–86

`esc_html_e()` function, [264](#)

`esc_html_x()` function, [266](#)–267

`esc_js()` function, [88](#)
`esc_sql()` function, [90](#)
`esc_url()` function, [84](#)
`esc_url_raw()` function, [85](#), [382](#)

Events Calendar, [157](#)–158

`_ex()` function, [265](#)–266

examples

custom action hooks, [128](#)–129

custom filter hooks, [129](#)

plugin, [4](#)–5

transients, [102](#)–103

executing cron, [236](#)

expanding, with additional field types, [52](#)–59

`$expire` parameter, [98](#)

F

Facebook, [146](#)

feedback, improving on validation errors, [51](#)–52

fields, adding to existing pages, [59](#)–62

`$fields` parameter, [320](#)

`<figure>` tag, [89](#)

File Transfer Protocol (FTP), [429](#)

files

deploying with FTP, SFTP, and SSH, [429](#)

naming, [28](#)

organizing, [15](#)–16

filter hooks

- about, [117–119](#)
 - commonly used, [124–126](#)
 - defined, [106](#)
 - functions for, [119–123](#)
 - quick return functions, [123–124](#)
 - filtering requests and responses, [304–306](#)
 - Firefox Developer Tools, [427](#)
 - `first_name` argument, [209](#)
 - `flush_rewrite_rules()` function, [390](#)
- ### folders
- structure of, [16–17](#)
 - using, [14](#)
 - `fopen()`, [293](#)
 - `force:balance:tags()` function, [86](#)
 - forcing balanced tags, [86](#)
 - `foreach` loop, [347](#)
 - form fields, [67–68](#)
 - `form()` method, [371](#), [372–373](#)
 - `<form>` element, [76](#)
 - `$format` parameter, [92](#), [93](#)
 - formatting SQL statements, [90–97](#)
 - Formatting toolbar, [155](#)
 - `formatting.php` file, [418](#)
 - forms, rendering, [47](#)
 - `form-table` class, [67–68](#)

forums, [224](#), [229](#), [424](#)–425
framework, for plugins, [13](#)–30
`fsockopen()`, [293](#), [294](#)
FTP (File Transfer Protocol), [429](#)
`function_exists()` function, [400](#)–401
functions. *See also specific functions*
 about, [3](#)
 of action hooks, [108](#)–113
 anonymous, [127](#)–128
 for filter hooks, [119](#)–123
 finding, [417](#)–418
 Multisite, [319](#)–321, [319](#)–350
 naming, [27](#)
 plugin activation, [22](#)–23
 plugin deactivation, [23](#)
 quick return, [123](#)–124
 Settings API, [43](#)–48
`functions.php` file, [418](#)
`$function_to_add` parameter, [107](#), [118](#)
`$function_to_remove` parameter, [109](#), [119](#)

G

generic queries, [96](#)
Genre taxonomy, registering, [193](#)–194
GET method, [280](#)
`$get_all` parameter, [320](#)
`get_all_category_ids()` function, [406](#)–407

get_avatar() function, [419](#)
get_blog_count() function, [350](#)
get_blog_details() function, [332](#), [351](#)
get_blog_option() function, [342](#), [349](#)
get_blog_post() function, [319](#)–320
get_blogs_of_user() function, [347](#)
_get_cron_array() function, [243](#)
get_current_user_id() function, [420](#)
get_network_option() function, [342](#)
get_networks() function, [352](#)
get_option() function, [36](#), [38](#)–39, [40](#), [44](#), [54](#)
get_pages() function, [419](#)
get_post() function, [14](#), [408](#)
get_post_meta() function, [185](#)–186, [191](#)
get_posts() function, [327](#), [419](#)
get_post_types() function, [419](#)
get_role() function, [230](#)–231
get_site_meta() function, [343](#)
get_sites() function, [321](#)
get_sitestats() function, [350](#)
get_submit_button() function, [47](#)
get_super_admins() function, [348](#)
get_taxonomy() function, [196](#)
get_the_author_description() function, [409](#), [417](#)
get_the_author_meta() function, [417](#)
get_the_term_list() function, [196](#)–197

get_transient() function, [102](#)
get_url_in_content() function, [418](#)
get_user_by() function, [349](#)
get_user_count() function, [350](#)
get_userdata() function, [214](#), [419](#)
get_user_meta() function, [218](#)–219
get_users() function, [207](#)
global (Multisite), [316](#)
GlotPress, [276](#)
GNU General Public License (GPL), [18](#)–19
GNU Gettext, [276](#)
Google Chrome DevTools, [428](#)
Google tag, [5](#)
GPL (GNU General Public License), [18](#)–19
grant_super_admin() function, [348](#)
\$group parameter, [98](#)
Gutenberg
 about, [149](#)–150
 Block Directory, [173](#)–174
 components of, [151](#)–155
 create-guten-block toolkit, [171](#)–173
 “Hello World!” block, [164](#)–168
 links, [151](#)
 technology stack of, [159](#)–164
 WP-CLI scaffolding, [168](#)–170
Gutenberg, Johannes (inventor), [150](#)

H

<h1> tag, [64](#), [87](#)

\$handle parameter, [273](#)

has_action() function, [111](#)–112

has_filter() function, [122](#)

have_posts() function, [323](#), [355](#)–356

<head> tag, [106](#)

header, plugin, [17](#)–19

header file, [18](#)

headers parameter, [296](#), [302](#)

headings, [64](#)–65

Heartbeat API

about, [395](#)

plugin, [396](#)–398

processing responses, [396](#)

receiving data, [395](#)–396

responding to data, [395](#)–396

sending data, [395](#)

using, [395](#)–398

"Hello World!" block, [164](#)–168

heredoc syntax, [306](#)

home_url() function, [21](#)

\$hook parameter, [236](#), [240](#)

hooks

- about, [74](#), [105](#)–106
- actions, [106](#)–117
- creating custom, [128](#)–129
- filters, [117](#)–126
- finding, [129](#)–131
- using from within classes, [126](#)–127
- using with anonymous functions, [127](#)–128
- variable, [130](#)

hooks database, [423](#)–424

HTML

- about, [86](#)
- escaping, [85](#)–86
- forcing balanced tags, [86](#)
- sanitizing, [86](#)–87
- URLs in, [84](#)

HTTP API, [2](#), [289](#)–294

HTTP API Handbook, [313](#)

HTTP extension, [292](#)–293

HTTP functions, [294](#)–306

HTTP requests, [289](#)–294

I

ID, Multisite, [319](#)

ID argument, [209](#), [213](#)

\$id parameter, [188](#), [196](#), [212](#), [214](#)

identical comparison operator (==), [111](#)

`if` statement, [355](#)
improving feedback on validation errors, [51](#)–52
Inactive status, for plugins, [10](#)
includes, [169](#)
`includes_url()` function, [21](#)
indentation, [28](#)–29
`init` action hook, [74](#), [114](#)–115, [134](#), [206](#), [359](#)
`init()` method, [127](#)
inline scripts, [137](#)–139
input data, validating/sanitizing, [81](#)
`insert()` method, [91](#)
Insomnia, [283](#)
installing plugins, [8](#)
`$instance` parameter, [371](#)–372
integers, sanitizing/validating, [81](#)–82
integrating non-WordPress pages, [391](#)–394
intention, authority vs., [74](#)–75

internationalization

about, [259](#)
creating translation files, [275](#)–278
echoing strings, [262](#)–271
of JavaScript, [273](#)–275
localization and, [260](#)–275
preparing plugins for translation, [261](#)–262
in professional work, [261](#)
reasons for, [260](#)
returning strings, [262](#)–271
using placeholders, [271](#)–272
`intval()` function, [82](#)
`is_archive()` function, [136](#)
`is_array()` function, [332](#)
`is_callable()` function, [401](#)
`is_email()` function, [84](#), [418](#)
`is_multisite()` function, [319](#), [322](#), [327](#)
`is_php_version_compatible()` function, [418](#)
`isset()` function, [308](#), [382](#), [409](#)
`is_singular()` function, [136](#)
`is_super_admin()` function, [227](#)–228, [348](#)
`is_tax()` function, [198](#)–199
`is_taxonomy_hierarchical()` function, [198](#)
`is_user_logged_in()` function, [206](#)–207
`is_user_member_of_blog()` function, [344](#)
`is_wp_version_compatible()` function, [418](#)

J

JavaScript

about, [88](#), [133](#)–134, [160](#)–161, [164](#)–165

Backbone, [146](#)

bundled scripts, [139](#)–141

custom scripts, [141](#)–146

enqueueing scripts, [135](#)–136

inline documentation, [417](#)

inline scripts, [137](#)–139

internationalizing, [273](#)–275

limiting scope, [136](#)

localizing scripts, [136](#)–137

polyfills, [141](#)

React, [146](#)–147

registering scripts, [134](#)–135

Underscore, [146](#)

jQuery, [142](#)–143

jQuery object, [143](#)–144

jQuery UI Sortable, [139](#)–140

JSON Formatter extension, [281](#)

JSON Web Token Authentication (JWT), [286](#)

`json_decode()` function, [297](#), [308](#)

JSX, [163](#)

JWT (JSON Web Token Authentication), [286](#)

K

KBabel, [276](#)

\$key parameter, [98](#)

key strings, [83](#)–84

Kinsta Blog, [313](#)

Know the Code, [426](#)

KSES script, [86](#)–87

L

last_name argument, [209](#)

Launchpad, [276](#)

layered system constraint, [280](#)

license, plugin, [18](#)–19

line break (\n), [83](#)

Link Category taxonomy, [192](#)

LinkedIn Learning, [427](#)

links, Gutenberg, [151](#)

load_plugin_textdomain() function, [261](#)–262

local events, [427](#)

local paths, [19](#)–20

localization, [136](#)–137, [260](#)–275

log file, [411](#)

loop, [355](#)–358

M

machine object (MO) files, [276](#)

"magic quotes," [80](#)

Make Network, [403](#)

Make WordPress Core site, [425](#)

Make WordPress section, [424](#)

managing plugins, [9](#)

`map_meta_cap()` function, [225](#)

Mature status (Multisite), [316](#)

media, as REST API endpoint, [282](#)

menus, adding, [31](#)–36

messages, [65](#)–66

meta boxes

about, [187](#)

adding custom, [188](#)–189

saving data, [190](#)–191

meta capabilities, [181](#), [225](#)

`$meta` parameter, [337](#)

`$meta_key` parameter, [183](#), [184](#), [185](#), [186](#), [187](#), [218](#), [219](#), [220](#)

`$meta_value` parameter, [184](#), [186](#), [187](#), [218](#), [219](#), [220](#)

`method` parameter, [296](#)

`method_exists()` function, [401](#)

methods

all-in-one, [91](#)–93

common, [93](#)–97

naming, [27](#)–28

uninstall, [24](#)–25

MO (machine object) files, [276](#)

Modern Tribe, [157](#)–158

`mod_rewrite` module, [385](#)

Multisite

about, [315](#)–316
advantages of, [317](#)
database schema, [350](#)–351
enabling in WordPress, [317](#)–318
functions, [319](#)–350
object classes, [352](#)
query classes, [351](#)–352
terminology, [316](#)–317
Multisite-specific tables, [350](#)–351
mu-plugins directory, [9](#)–10
Must-Use status, for plug-ins, [10](#)
mysql_query() function, [90](#)–91

N

\n (line break), [83](#)
_n() function, [267](#)–268
\$name argument, [214](#)
\$namespace parameter, [287](#)
namespaces, [14](#)–15, [287](#)
Nav menu item, as content, [176](#)
Nav Menu taxonomy, [192](#)
NetBeans IDE, [428](#)

network (Multisite)

content shortcode examples, [325](#)–330

content widget example, [330](#)–336

defined, [316](#)

options for, [342](#)–343

stats, [350](#)

`$new_instance` parameter, [374](#)

Nginx, [318](#), [403](#)

`nickname` argument, [209](#)

`_n_noop()` function, [269](#)–270

`noconflict()`, [145](#)

no-conflict mode (WordPress), [144](#)

Node.js, [162](#)

nonces

about, [74](#)

authority vs. intention, [74](#)–75

creating, [75](#)–78

verifying, [75](#)–78

non-WordPress pages, integrating, [391](#)–394

Notepad++, [428](#)

`$number` parameter, [267](#)

`_nx()` function, [268](#)–269

`_nx_noop()` function, [270](#)–271

O

OAuth 1.0a Server, [286](#)

object classes, [352](#)

`$object_type` parameter, [193](#), [195](#)

`$old_instance` parameter, [374](#)

`*_option()` function, [418](#)

Options API

about, [2](#), [36](#), [102](#)

`autoload` parameter, [40](#)–42

deleting options, [40](#)

loading arrays of options, [39](#)–40

retrieving options, [38](#)–39

saving arrays of options, [37](#)

saving options, [36](#)–37

updating options, [37](#)–38

`$output` parameter, [94](#)

`$override` parameter, [287](#)

P

page revision, as REST API endpoint, [282](#)

`page-numbers` class, [70](#)

pages

as content, [176](#)

as REST API endpoint, [282](#)

pagination, [69](#)–70

`$pairs` parameter, [362](#)

parameters, building shortcodes with, [362](#)–364

`parse_request()` function, [387](#)

PascalCase, [27](#)

`$path` parameter, [336](#)

paths, [19–21](#)
paths, plugin, [19](#)
pdev2, [168–169](#)
pdev_advanced_dashboard_control() function, [380](#), [381](#)
pdev_advanced_dashboard_display() function, [380](#)
pdev_book_collection_post_types() function, [179](#)
pdev_book_details_meta_box() function, [189](#)
pdev_comment_days_field() function, [255](#)
pdev_comments_status_field() function, [255](#)
pdev_create_menu() function, [32](#)
pdev_cron_activation() function, [237](#)
pdev_cron_schedules() function, [242](#)
pdev_example_text hook, [129](#)
pdev_load_after action hook, [129](#)
pdev_movie_api_results() function, [297](#)
pdev_multisite_sort_posts_array() function, [328](#)
pdev_multisite_switch_page() function, [322](#)
pdev_plugin_admin_init() function, [52–53](#)
pdev_plugin_option_page() function, [35](#)
pdev_plugin_setting_beast_mode() function, [54](#)
pdev_plugin_setting_fav_holiday() function, [53–54](#)
pdev_plugin_validate_options() function, [44](#), [46](#)
pdev_return_post_title_by_author_id() function, [287](#)
pdev_simple_dashboard_display() function, [378](#)
pdev_styling_settings() function, [63](#)
performance, [97](#)

permalinks, [384](#)–385, [391](#)–394

PHP

about, [161](#)–162, [164](#), [403](#)

HTTP requests in, [292](#)–294

inline documentation, [416](#)–417

shorthand, [30](#)

PHP Framework Interprop Group (PHP-FIG), [26](#)

PHPDoc, [26](#)–27, [416](#)

PHP-FIG standards, [15](#), [17](#)

phpMyAdmin, [429](#)

PhpStorm, [428](#)

PHPXref, [422](#)–423

placeholders, [271](#)–272

pluggable functions, [3](#)

pluggable.php file, [419](#)

plugin activation function, [22](#)–23

plugin Administration page, [43](#)

Plugin API, [2](#), [105](#)–106. *See also* [hooks](#)

plugin deactivation function, [23](#)

Plugin Developer Handbook, [275](#), [420](#)–421

Plugin Directory, [4](#), [14](#), [155](#)

plugin header, [17](#)–19

plugin_dir_*() function, [419](#)

plugin_dir_path() function, [19](#)–20

plugin_dir_url() function, [20](#)–21

plugin.php file, [16](#), [419](#)

\$plugin_rel_path parameter, [262](#)

#pluginreview, [425](#)

plugins

about, [1](#)
activating functions, [21](#)–23
advantages of, [5](#)–7
available, [4](#)–5
best practices, [14](#)–17
Blog Pester, [247](#)–250
caching data within, [100](#)–101
coding standards, [25](#)–30
creating nonces in, [77](#)–78
creating with user metadata, [220](#)–223
custom role and capability, [231](#)–233
deactivating functions, [21](#)–23
debugging and, [401](#)–402
Delete Comments, [253](#)–258
determining paths, [19](#)–21
directories for, [9](#)–10
downloading, [4](#)
editing, [9](#)
Events Calendar, [157](#)–158
examples of, [4](#)–5, [155](#)–159
framework for, [13](#)–30
Heartbeat API, [396](#)–398
installing, [8](#)
loading, [3](#)
managing, [8](#)–11

naming, [14](#)
plugin header, [17](#)–19
post metadata, [199](#)–204
Post Type Switcher, [158](#)–159
post types, [199](#)–204
preparing for translation, [261](#)–262
requirements for, [13](#)–14
reusing, [7](#)
segregating options for, [41](#)
separating from themes, [6](#)–7
settings, [36](#)
sharing, [7](#)
tags for, [5](#)
taxonomies, [199](#)–204
types of, [10](#)–11
uninstall methods, [24](#)–25
updating, [7](#)
uses for, [389](#)
verifying nonces in, [77](#)–78
WooCommerce, [156](#)–157
WordPress interactions with, [2](#)–3
WP-CLI scaffolding, [168](#)–169
Plugins team, [424](#)
plugins_loaded action hook, [113](#)–114, [127](#), [134](#)
\$plural parameter, [267](#)
PO (portable object) files, [276](#)

Poedit, [276](#)
polyfills, [141](#)
Polyglots team, [424](#)
portable object (PO) files, [276](#)
portable object template (POT) file, [276](#), [277](#)
post metadata
 about, [183](#)
 adding, [184](#)–185
 deleting, [186](#)–187
 plugins, [199](#)–204
 registering, [183](#)–184
 retrieving, [185](#)–186
 updating, [186](#)
POST method, [280](#)
\$post parameter, [190](#)
post revisions, as REST API endpoint, [282](#)
Post Status, [426](#)
post statuses, as REST API endpoint, [283](#)
Post Tag taxonomy, [192](#)
Post Type Switcher, [158](#)–159

post types

- assigning taxonomies to, [194](#)–195
- attaching taxonomies, [182](#)–183
- creating custom, [176](#)–183
- plugins, [199](#)–204
- possibilities of, [176](#)–177
- registering, [177](#)–179
- as REST API endpoint, [282](#)
- setting labels, [179](#)–180
- using custom capabilities, [180](#)–182

`$post_id` parameter, [184](#), [185](#), [186](#), [187](#), [190](#), [320](#)

Postman, [283](#)–284

- `*_post_meta()` function, [419](#)
- `post.php` file, [419](#)

posts

- as content, [176](#)
- displaying, [353](#)–358
- querying, [353](#)–358
- as REST API endpoint, [282](#)
- using taxonomies with, [196](#)–197

`$post_type` parameter, [183](#), [215](#)

POT (portable object template) file, [276](#), [277](#)

`$pref` parameter, [341](#)–342

`preg_match()` function, [83](#)–84

`$prev_value` parameter, [186](#), [219](#)

primitive capabilities, [225](#)

`printf()` function, [267](#), [271](#)–272
\$priority parameter, [107](#), [109](#)–110, [112](#), [118](#), [119](#), [120](#), [122](#), [188](#)
proxy support, [303](#)–304
/public file, [16](#)
Public status (Multisite), [316](#)
\$public_only parameter, [215](#)
PUT method, [280](#)
PuTTY, [284](#), [429](#)

Q

Quantcast, [142](#)
queries
database, [89](#)–90
generic, [96](#)
posts, [353](#)–358
process of, [386](#)–389
protecting against SQL injections, [96](#)–97
WordPress and, [386](#)
query classes, [351](#)–352
Query Monitor, [411](#)–414
\$query parameter, [94](#)
quotes, single/double, [28](#)

R

React, [141](#), [146](#)–147, [163](#)
\$reassign parameter, [212](#), [346](#)
Recently Active status, for plugins, [10](#)

\$recurrence parameter, [236](#)
recurring events, scheduling, [236](#)–238
redirects, URLs in, [85](#)
reference lists, for hooks, [130](#)–131
\$refresh parameter, [341](#)–342
`register_activation_hook()` function, [22](#), [113](#), [419](#)
`register_deactivation_hook()` function, [23](#), [113](#), [419](#)
registering
 custom shortcodes, [359](#)–366
 custom taxonomies, [192](#)–194
 Genre taxonomy, [193](#)–194
 post metadata, [183](#)–184
 post types, [177](#)–179
 scripts, [134](#)–135
 settings, [44](#)
`register_post_meta()` function, [183](#)–184
`register_post_type()` function, [177](#), [179](#), [180](#), [181](#), [182](#), [194](#)–195, [419](#)
`register_rest_route()` function, [287](#)
`register_setting()` function, [44](#), [46](#), [47](#), [60](#)
`register_taxonomy()` function, [193](#), [194](#)–195, [196](#)
`register_taxonomy_for_object_type()` function, [195](#)
`register_uninstall_hook()` function, [25](#), [62](#), [419](#)
`register_widget()` function, [369](#)
regular expressions, [46](#), [84](#)
`remove_action()` function, [108](#)–109, [109](#)–110, [111](#)
`remove_all_actions()` function, [109](#)–110

`remove_all_filters()` function, [120](#)–121

`remove_all_shortcodes()` function, [367](#)

`remove_cap()` function, [231](#)

`remove_filter()` function, [119](#)–120, [122](#)

`remove_role()` function, [230](#)

`remove_shortcode()` function, [367](#)

`remove_user_from_blog()` function, [346](#)

removing

capabilities from roles, [231](#)

settings, [62](#)–63

rendering forms, [47](#)

Representational State Transfer (REST). *See* [REST API](#)

request/response protocol, [289](#)

requests, filtering, [304](#)–306

/resources file, [16](#)

response parameter, [302](#)

responses

to data, [395](#)–396

filtering, [304](#)–306

processing, [396](#)

REST API

about, [2](#), [279](#)–280
accessing WordPress REST API, [281](#)–283
clients, [283](#)–289
HTTP API, [289](#)–294
uses for, [280](#)
using, [307](#)–313
WordPress HTTP functions, [294](#)–306
REST API Authentication, [286](#)
REST API Handbook, [313](#)
`restore_current_blog()` function, [321](#)–323, [324](#), [326](#), [327](#), [333](#)
restoring Multisite sites, [321](#)–324
`rest_url()` function, [21](#)
results, selecting generic, [95](#)–96
`_return_empty_array()` function, [124](#)
returning strings, [262](#)–271
reusable block, as content, [176](#)
reusing plugins, [7](#)
revision, as content, [176](#)
`revoke_super_admin()` function, [348](#)

Rewrite API

about, [2](#), [384](#)

Apache mod_rewrite module, [385](#)

permalink principles, [384](#)–385

practical uses for, [389](#)–394

queries, [386](#)–389

URLs, [384](#), [385](#)–386

\$role parameter, [228](#), [344](#)

roles and capabilities

about, [223](#)

adding capabilities to roles, [230](#)–231

creating roles, [228](#)–230

custom plugin for, [231](#)–233

custom roles, [224](#)

customizing roles, [228](#)–233

default roles, [224](#)

deleting roles, [230](#)

removing capabilities from roles, [231](#)

users and, [343](#)–347

\$route parameter, [287](#)

rows, selecting, [94](#)

S

sanitize_callback argument, [184](#)

sanitize_email() function, [84](#)

sanitize_html_class() function, [89](#)

sanitize_text_field() function, [55](#), [82](#)–83, [374](#)

sanitizing

data (See [data sanitization](#))

HTML, [86](#)–87

integers, [81](#)–82

`save_post` action hook, [116](#), [130](#), [190](#)

saving

arrays of options, [37](#)

cached data, [98](#)–99

expiring options, [102](#)

meta box data, [190](#)–191

saving options, [36](#)–37

scheduled tasks

about, [235](#)

cron, [235](#)–236

practical use, [247](#)–258

scheduling cron events, [236](#)–246

true cron, [247](#)

scheduling cron events, [236](#)–246

scope, limiting, [136](#)

`$screen` parameter, [188](#)

`<script>` tag, [87](#)

scripts

 bundled (*See* [bundled scripts](#))

 custom (*See* [custom scripts](#))

 enqueueing, [135](#)–136

 inline, [137](#)–139

 localizing, [136](#)–137

 registering, [134](#)–135

 search, as REST API endpoint, [283](#)

sections

 adding, [59](#)–60

 defining, [44](#)–46

Secure File Transfer Protocol (SFTP), [8](#), [429](#)

Secure Shell (SSH), [429](#)

security

 about, [72](#)

 data validation/sanitzation, [78](#)–90

 good habits for, [97](#)

 nonces, [74](#)–78

 user permissions, [72](#)–74

selected() function, [54](#)

self-closing shortcodes, [364](#)–366

\$sep parameter, [196](#)

server variables, [88](#)

set_post_thumbnail() function, [419](#)

settings

- about, [31](#)
- defining, [44](#)–46
- log file location, [411](#)
- plugins, [36](#)
- post type labels, [179](#)–180
- registering, [44](#)
- removing, [62](#)–63
- as REST API endpoint, [283](#)

Settings API

- about, [2](#), [42](#)
- benefits of, [42](#)–43
- creating plugin Administration page, [43](#)
- defining sections/settings, [44](#)–46
- functions of, [43](#)–48
- registering settings, [44](#)
- rendering forms, [47](#)
- validating user input, [46](#)–47
- `settings_fields()` function, [47](#)
- SFTP (Secure File Transfer Protocol), [429](#)
- sharing plugins, [7](#)
- Shortcode API, [2](#)
- `$shortcode` parameter, [362](#)
- `shortcode_atts()` function, [362](#), [364](#)

shortcodes

- about, [358](#)–359
 - building, [360](#)–366
 - building with content, [364](#)–366
 - building with parameters, [362](#)–364
 - network content examples, [325](#)–330
 - registering custom, [359](#)–366
 - self-closing, [364](#)–366
 - tips for, [366](#)–367
- shorthand tags, [30](#)
- show_in_rest argument, [184](#)
 - sidebar, [155](#)
 - Sidebar menu, [155](#)
 - \$sidebar parameter, [371](#)–372
 - Simple Mail Transfer Protocol (SMTP), [3](#)
 - single argument, [184](#)
 - \$single parameter, [185](#), [218](#), [267](#)
 - single quotes, [28](#)
 - site meta, [343](#)
 - \$site_id parameter, [214](#), [320](#), [337](#), [341](#)–342, [344](#), [346](#)

sites (Multisite)

- checking owners, [349](#)–350
- creating, [336](#)–342
- defined, [316](#)
- options for, [342](#)
- restoring, [321](#)–324
- switching, [321](#)–324
- site-specific tables, [351](#)
- site_url() function, [21](#)
- Slack, [425](#)
- SMTP (Simple Mail Transfer Protocol), [3](#)
- software, WordPress Core, [3](#)
- space usage, [29](#)–30
- spam status (Multisite), [316](#)
- tag, [65](#)
- sprintf() function, [271](#)–272
- SQL injections, protecting queries against, [96](#)–97
- SQL statements, [30](#), [90](#)–97
- /src file, [16](#)
- SSH (Secure Shell), [429](#)
- stateless constraint, [280](#)
- statements, SQL, [30](#)

strings

echoing, [262](#)–271

email, [84](#)

key, [83](#)–84

returning, [262](#)–271

`strip_shortcodes()` function, [367](#)

`strip_tags()` function, [83](#)

`` tag, [86](#)–87

`strtotime()` function, [248](#)–249

structure, of folders, [16](#)–17

Sublime Text, [428](#)–429

`submenuPage()` method, [243](#)

submenus, adding, [31](#)–36

`submit_button()` function, [47](#), [66](#)

subscriber role, [207](#)

Subscriber role, [224](#)

Super Admin role, [224](#), [348](#)

support forums, [424](#)–425

`switch_to_blog()` function, [321](#)–323, [324](#), [326](#), [333](#)

syntax, jQuery, [143](#)

T

`$table` parameter, [92](#), [93](#)

tables, [68](#)–69

`$tag` parameter, [106](#), [107](#), [109](#)–110, [117](#), [118](#), [119](#), [120](#), [121](#), [122](#), [359](#)

tags

- for plugins, [5](#)
- as REST API endpoint, [282](#)
- shorthand, [30](#)

taxonomies

- about, [175](#), [191](#)–192
- assigning to post types, [194](#)–195
- attaching, [182](#)–183
- conditional tags, [197](#)–199
- creating custom, [191](#)–195
- plugins, [199](#)–204
- registering custom, [192](#)–194
- as REST API endpoint, [282](#)
- retrieving, [196](#)
- using custom, [195](#)–199
- using with posts, [196](#)–197
- \$taxonomy parameter, [193](#), [195](#), [196](#), [197](#), [198](#)
- taxonomy_exists() function, [197](#)
- template() method, [243](#), [244](#)
- template_include filter hook, [125](#)–126
- \$term parameter, [198](#)
- \$text parameter, [263](#), [265](#)
- text strings, arbitrary, [82](#)–83
- TextMate, [428](#)
- tfoot tag, [69](#)
- thead tag, [69](#)

the_content filter hook, [124](#)–125

Theme Customization (Customize) API, [3](#)

themes

as REST API endpoint, [283](#)

separating from plugins, [6](#)–7

the_post() function, [323](#), [355](#)–356

the_terms() function, [196](#)

the_title() function, [357](#)

timeout parameter, [296](#)

\$timestamp parameter, [236](#), [240](#)

TinyMCE, [149](#)

\$title parameter, [188](#), [337](#)

tools

browsers, [427](#)–428

Codex, [421](#)–422

community resources, [424](#)–427

core WordPress code, [415](#)–420

deploying files, [429](#)

editors, [428](#)–429

phpMyAdmin, [429](#)–430

Plugin Developer Handbook, [420](#)–421

websites for, [422](#)–424

top-level menus, creating, [32](#)–33

Transient API, [2](#), [97](#). *See also* [transients](#)

transients

- about, [101–102](#)
- deleting expiring options, [102](#)
- example of, [102–103](#)
- retrieving expiring options, [102](#)
- saving expiring options, [102](#)
- technical details, [103](#)

translation

- creating files, [275–278](#)
- preparing plugins for, [261–262](#)
- tools for, [276](#)

Twitter, [427](#)

Twitter tag, [5](#)

type argument, [183](#)

U

Underscore, [146](#)

Undo button, [9](#)

uniform interface constraint, [279](#)

uninstall methods, [24–25](#)

`uninstall.php` file, [16, 24–25](#)

`$unique` parameter, [184, 218](#)

`unregister_setting()` function, [62–63](#)

unscheduling events, [240–241](#)

`update()` method, [91, 371, 373–374](#)

`$update` parameter, [190](#)

`update_blog_option()` function, [342](#)

update_blog_status() function, [341](#)
update_network_option() function, [342](#)
update_option() function, [36](#), [37](#)–38, [41](#)
update_post_meta() function, [186](#)
update_site_meta() function, [343](#)
update_user_meta() function, [219](#)–220

updating

- options, [37](#)–38
- plugins, [7](#)
- post metadata, [186](#)
- posts using REST API, [309](#)
- user metadata, [219](#)–220
- users, [208](#)–209, [211](#)–212

URL paths, [20](#)–21

URLs, [84](#)–85, [384](#), [385](#)–386

user data

- about, [205](#)–206
- customizing roles, [228](#)–233
- getting, [213](#)–214
- getting user objects, [213](#)–215
- getting user post counts, [215](#)–217
- limiting access, [225](#)–228
- roles and capabilities, [223](#)–224
- user metadata, [217](#)–223
- working with, [206](#)–223

user input, validating, [46](#)–47

user interface (UI), [62](#), [64](#)

user metadata

about, [217](#)

adding, [218](#)

creating plugins with, [220](#)–223

deleting, [220](#)

retrieving, [218](#)–219

updating, [219](#)–220

user objects, getting, [213](#)–215

user permissions, [72](#)–74, [225](#)–227

`user_agent` parameter, [296](#)

`$userdata` parameter, [209](#)

`user_email` argument, [209](#), [213](#)

`$user_id` parameter, [215](#), [218](#), [219](#), [220](#), [337](#), [344](#), [346](#)

`user_login` argument, [209](#), [213](#)

`*_user_meta()` function, [420](#)

`username_exists()` function, [210](#)

`user_nicename` argument, [209](#), [213](#)

`user_pass` argument, [209](#), [213](#)

`user.php` file, [420](#)

`user_registered` argument, [209](#), [213](#)

users

about, [205](#)–206
creating, [208](#)–211
customizing roles, [228](#)–233
deleting, [208](#)–209, [212](#)–213
limiting access, [225](#)–228
as REST API endpoint, [282](#)
roles and capabilities, [223](#)–224, [343](#)–347
updating, [208](#)–209, [211](#)–212
user functions, [206](#)–208
working with, [206](#)–223
`user_url` argument, [209](#), [213](#)
`usort()` function, [328](#)

V

validating

data (See [data validation](#))
improving feedback on errors, [51](#)–52
integers, [81](#)–82
user input, [46](#)–47
`$value` parameter, [117](#), [341](#)–342
Vanilla JavaScript, [161](#)
variable hooks, [130](#)

variables

- environment, [88](#)
- naming, [27](#)
- selecting, [93](#)
- server, [88](#)

version support, [400](#)–401

Visual Studio Code, [429](#)

Vue, [146](#)

W

webpack, [163](#), [165](#)–166

websites. *See also specific websites*

- coding standards, [25](#)
- community news, [426](#)–427
- hook references lists, [130](#)–131
- hooks database, [423](#)–424
- regular expressions, [84](#)
- support forums, [424](#)–425
- for tools, [421](#)–424

wget command, [247](#)

\$where parameter, [92](#)

\$where_format parameter, [92](#)

while loop, [355](#)

whitelist validation, [80](#)

whitelisting, [43](#)

widefat class, [68](#)–69

Widget API, [2](#)

`widget()` method, [371](#)
`$widget_id` parameter, [378](#)
`$widget_name` parameter, [378](#)
widgets, [330](#)–336, [368](#)–384
`widgets_init` action hook, [330](#), [369](#)
WooCommerce, [4](#), [156](#)–157, [402](#)
WordCamp, [427](#)
WordPress. *See also* *specific topics*
 development of, [402](#)–404
 enabling Multisite in, [317](#)–318
 HTTP functions, [294](#)–306
 interactions with plug ins, [2](#)–3
 no-conflict mode, [144](#)
 queries and, [386](#)
 React and, [146](#)–147
 version support, [400](#)–401
WordPress Core software, [3](#)
WordPress Ideas, [426](#)
WordPress News, [426](#)
WordPress Planet, [426](#)
WordPress REST API. *See* [REST API](#)
WordPress Slack, [425](#)
WordPress toolbar, [73](#)
WordPress UI, [64](#)
WP CLI, [276](#), [278](#)
`wp` JavaScript global, [140](#)–141

`wp_add_dashboard_widget()` function, [378](#), [380](#)
`wp_add_inline_script()` function, [137](#)–138
`wp.ally.speak()`, [140](#)
`wp_cache_add()` function, [98](#)–99
`wp_cache_delete()` function, [99](#)
`wp_cache_get()` function, [99](#)
`wp_cache_replace()` function, [98](#)–99
`wp_cache_set()` function, [98](#)–99
WP-CLI scaffolding, [168](#)–170
`wp_create_user()` function, [211](#)
`$wpdb` object, [90](#)–91
`$wpdb->get_col()` method, [94](#)–95
`$wpdb->get_results()` method, [95](#)–96
`$wpdb->get_row()` method, [94](#)
`$wpdb->get_var()` method, [93](#)
`$wpdb->insert()` method, [92](#)–93
`$wpdb->prepare()` method, [96](#)–97
`$wpdb->query()` method, [96](#)
`$wpdb->update()` method, [91](#), [92](#)
`wp_delete_attachment()` function, [419](#)
`wp_delete_user()` function, [212](#)–213
WP_Dependencies API, [135](#)
`wp_enqueue_script()` function, [117](#), [274](#)
`wp.escapeHTML`, [140](#)
`wp_filter_nohtml_kses()` function, [81](#)
WPForms, [4](#)

`wp_get_current_user()` function, [73](#), [215](#), [419](#)
`wp_get_schedules()` function, [243](#)
`wp_head` action hook, [116](#)–117
`wp_head()` function, [116](#)
`wp.heartbeat`, [141](#)
`wp.i18n`, [140](#)–141
`wp_insert_user()` function, [209](#)–211
`wp_json_encode()` function, [88](#)
`wp_kses()` function, [86](#)–87
`wp_kses_data()` function, [87](#)
`wp_list_categories()` function, [198](#)
`wp_localize_jquery_ui_datepicker()` function, [137](#)
`wp_localize_script()` function, [137](#), [273](#), |274–275
`wp_mail()` function, [3](#), [6](#), [249](#), [419](#)
`wpmu_create_blog()` function, [338](#)
`wp_nav_menu()` function, [324](#)
`WP_Network`, [352](#)
`WP_Network_Query`, [352](#)
`wp_next_scheduled()` function, [237](#), [240](#)–241
`wp_nonce:*`() function, [418](#)
`wp_nonce_field()` function, [76](#)
`wp_nonce_url()` function, [76](#)
`wp_*_post()` function, [419](#)
`WP_Query` class, [354](#)–355
`$wp_query` object, [388](#)
`wp_rand()` function, [419](#)

`wp_redirect()` function, [85](#), [419](#)
`wp_register_script()` function, [134](#)–135
`wp_remote_` functions, [294](#)–303
`wp_remote_get()` function, [297](#)
`wp_remote_post()` function, [308](#)
`wp_remote_request()` function, [295](#)
`wp_remote_retrieve_body()` function, [297](#)
`wp_remote_retrieve_response_message()` function, [308](#)
`wp_reset_postdata()` function, [356](#)
`$wp_rewrite` object, [387](#)–388
`wp_safe_redirect()` function, [85](#)
`wp_schedule_event()` function, [236](#)–237, [242](#)
`wp_schedule_single_event()` function, [238](#)–240
`WP_Site`, [352](#)
`WP_Site_Query`, [351](#)–352
`wp_strip_all_tags()` function, [83](#), [85](#), [418](#)
`wp_tag_cloud()` function, [198](#)
`wp_unschedule_event()` function, [240](#)–241
`wp_unslash()` function, [80](#)
`wp_update_user()` function, [211](#)–212
`wp_upload_dir()` function, [418](#)
`wp_*_user()` function, [420](#)
`WP_Widget` class, [368](#)
`wp_widget_rss_output()` function, [381](#)

X

`_x()` function, [264](#)–265

\$x parameter, [94](#)

Y

\$y parameter, [94](#)

Yoast SEO, [4](#)

Z

zip uploader, [8](#)

Copyright © 2020 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada and the United Kingdom

ISBN: 978-1-119-66694-3

ISBN: 978-1-119-66697-4 (ebk)

ISBN: 978-1-119-66693-6 (ebk)

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2020935186

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. WordPress is a registered trademark of WordPress Foundation Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

To my son, Lucas Royce Williams, for reminding me to always stay curious.
—BRAD WILLIAMS

*To my grandmother, Angeline “Granny” Frazier, for keeping me fed with
home-cooked meals during the writing process.*
—JUSTIN TADLOCK

*To my wife, Keri, for her boundless support and patience. To Brad and
Justin for inviting me along. To everyone who has helped me along the way.
To Penny and Paul.*
—JOHN JAMES JACOBY

ABOUT THE AUTHORS

BRAD WILLIAMS is the CEO and cofounder of WebDevStudios, one of the oldest and most respected WordPress agencies in the world. He is a coauthor of the Professional WordPress book series. Brad is the co-organizer of WordCamp Philly 2010–2015 and the Philadelphia WordPress Meetup. He was also a co-organizer for the Inaugural WordCamp US in 2015 and again in 2016. Brad has been developing websites for more than 20 years, including the last 10 where he has focused on open source technologies like WordPress.

JUSTIN TADLOCK began using WordPress in 2005 as a platform to share his writing. He fell in love with design and development, which prompted him to launch his first business in 2008. He ran Theme Hybrid, one of the earliest theme and plugin shops, for more than a decade. During this time, he created more than 70 plugins and themes for public release and many more for custom jobs. He also spent time as an administrator for the official WordPress theme review team. His latest adventure is writing for WP Tavern. Outside WordPress, Justin likes to collect films and makes time to read every day. You can follow his personal journey at justintadlock.com.

JOHN JAMES JACOBY is a software developer who started writing code at 13 years old, late at night, in a dark bedroom, on a shabby computer that his dad spent way too much money on. He found bbPress first, then BuddyPress, and finally WordPressMU, which is now simply known as WordPress Multisite. He's been the project lead for BuddyPress and bbPress since 2008; is a member of the WordPress.org Administration and Security teams and a component maintainer for Multisite, Options, and User Roles; and has contributed to more than 50 free plugins on WordPress.org that are active on nearly 15 million websites. When he's not coding, he likes spending time at home with his family, paddle boarding, and grumping around his local community. You can find him on the web at jjj.blog.

ACKNOWLEDGMENTS

TO MY SON, LUCAS, for being my daily inspiration and best friend. To the love of my life, April, thank you for your endless support, friendship, and continuing to put up with my nerdy ways. To Lisa Sabin-Wilson, your partnership and friendship have helped me grow not only as a business leader but as a person. To Justin and John, two amazing coauthors, your knowledge of WordPress is unmatched, and this book wouldn't have been what it is without you both. Thank you to the entire WordPress community for your support, friendships, motivation, and guidance over the years.

—BRAD WILLIAMS

To the WordPress community, you've been my online family for 15 years. You've given me an opportunity to have a career doing something I'm passionate about. It's been a wild ride thus far. To the thousands of users who've installed my plugins and themes, thank you for sticking with me all these years. To my parents, thank you for allowing me to dream big. To my family and friends, I'll try to find better dinner conversation topics than WordPress in the future. Thanks for lending an ear on a topic you know nothing about other than it being my job. To my cats, each day is a mixed bag of frustration and joy, but it's never boring. Thanks for mostly staying off my keyboard while I'm writing.

—JUSTIN TADLOCK

Thank you to Matt for his trust; Jen for calling me Jtrip; Jake for his words of wisdom; Andy, Boone, Paul, and Stephen for their leadership on the bbPress and BuddyPress plugins; Pippin for always putting people first; and everyone in the WordPress community for their support these past 13+ years. To my plugin users, thank you for your continued feedback to help make the little projects we enjoy as good as they can be. To my real-life friends, thank you for being the quirky and fun people who you are. To my WordPress family, you are the greatest. To my parents, who always did their best with what they had. To you, dear reader, for making it this far.

—JOHN JAMES JACOBY

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.