Plugin Handbook

Welcome to the WordPress Plugin Developer Handbook; are you ready to jump right in to the world of WordPress plugins?

The Plugin Developer Handbook is a resource for all things WordPress plugins. Whether you're new to WordPress plugin development, or you're an experienced plugin developer, you should be able to find the answer to many of your plugin-related questions right here.

- If you're new to plugin development, start by reading the introduction and then move on to the basics.
- The info in plugin security will introduce best practices for security related stuff.
- Hooks are what make your plugin interact with WordPress, and how you can let other developers interact with your plugin.
- Privacy will help you understand about handling sensitive data.
- To find out more about WordPress' built-in functionality that you can use in your plugin, check out Administration Menus, Shortcodes, Settings, Metadata, Custom Post Types, Taxonomies, and Users.
- Learn about getting data using the HTTP API.
- If you're using JavaScript, jQuery or Ajax in your plugin, you'll find the information you need in that section.
- To learn about time-based WordPress tasks, check out the Cron chapter.
- Internationalization is how you get your plugin ready for use in locales other than your own.
- When all that is done, you can prepare your plugin for inclusion in the Plugin Directory
- Finally: some developer tools you might find useful.

The WordPress Plugin Developer Handbook is created by the WordPress community, for the WordPress community. We are always looking for more contributors; if you're interested, stop by the docs team blog to find out more about getting involved.

First published

July 30, 2014

Last updated

December 14, 2023

Code is Poetry

Plugin Basics

Getting Started

At its simplest, a WordPress plugin is a PHP file with a WordPress plugin header comment. It's highly recommended that you create a directory to hold your plugin so that all of your plugin's files are neatly organized in one place.

To get started creating a new plugin, follow the steps below.

1. Navigate to the WordPress installation's **wp-content** directory.
2. Open the **plugins** directory.
3. Create a new directory and name it after the plugin (e.g. `plugin-name`).
4. Open the new plugin's directory.
5. Create a new PHP file (it's also good to name this file after your plugin, e.g. `plugin-name.php`).

Here's what the process looks like on the Unix command line:

```
wordpress $ cd wp-content
wp-content $ cd plugins
plugins $ mkdir plugin-name
plugins $ cd plugin-name
plugin-name $ vi plugin-name.php
```

In the example above, `vi` is the name of the text editor. Use whichever editor that is comfortable for you.

Now that you're editing your new plugin's PHP file, you'll need to add a plugin header comment. This is a specially formatted PHP block comment that contains metadata about the plugin, such as its name, author, version, license, etc. The plugin header comment must comply with the header requirements, and at the very least, contain the name of the plugin.

Only one file in the plugin's folder should have the header comment — if the plugin has multiple PHP files, only one of those files should have the header comment.

After you save the file, you should be able to see your plugin listed in your WordPress site. Log in to your WordPress site, and click **Plugins** on the left navigation pane of your WordPress Admin. This page displays a listing of all the plugins your WordPress site has. Your new plugin should now be in that list!

Hooks: Actions and Filters

WordPress hooks allow you to tap into WordPress at specific points to change how WordPress behaves without editing any core files.

There are two types of hooks within WordPress: *actions* and *filters*. Actions allow you to add or change WordPress functionality, while filters allow you to alter content as it is loaded and displayed to the website user.

Hooks are not just for plugin developers; hooks are used extensively to provide default functionality by WordPress core itself. Other hooks are unused place holders that are simply available for you to tap into when you need to alter how WordPress works. This is what makes WordPress so flexible.

Basic Hooks

The 3 basic hooks you'll need when creating a plugin are the register_activation_hook() , the register_deactivation_hook() , and the register_uninstall_hook() .

The activation hook is run when you *activate* your plugin. You would use this to provide a function to set up your plugin — for example, creating some default settings in the `options` table.

The deactivation hook is run when you *deactivate* your plugin. You would use this to provide a function that clears any temporary data stored by your plugin.

These uninstall methods are used to clean up after your plugin is *deleted* using the WordPress Admin. You would use this to delete all data created by your plugin, such as any options that were added to the `options` table.

Adding Hooks

You can add your own, custom hooks with do_action() , which will enable developers to extend your plugin by passing functions through your hooks.

Removing Hooks

You can also use invoke remove_action() to remove a function that was defined earlier. For example, if your plugin is an add-on to another plugin, you can use remove_action() with the same function callback that was added by the previous plugin with add_action() . The priority of actions is important in these situations, as remove_action() would need to run after the initial add_action() .

You should be careful when removing an action from a hook, as well as when altering priorities, because it can be difficult to see how these changes will affect other interactions with the same hook. We highly recommend testing frequently.

You can learn more about creating hooks and interacting with them in the Hooks section of this handbook.

WordPress APIs

Did you know that WordPress provides a number of Application Programming Interfaces (APIs)? These APIs can greatly simplify the code you need to write in your plugins. You don't want to reinvent the wheel, especially when so many people have done a lot of the work and testing for you.

The most common one is the Options API, which makes it easy to store data in the database for your plugin. If you're thinking of using cURL in your plugin, the HTTP API might be of interest to you.

Since we're talking about plugins, you'll want to study the Plugin API. It has a variety of functions that will assist you in developing plugins.

How WordPress Loads Plugins

When WordPress loads the list of installed plugins on the Plugins page of the WordPress Admin, it searches through the `plugins` folder (and its sub-folders) to find PHP files with WordPress plugin header comments. If your entire plugin consists of just a single PHP file, like Hello Dolly, the file could be located directly inside the root of the `plugins` folder. But more commonly, plugin files will reside in their own folder, named after the plugin.

Sharing your Plugin

Sometimes a plugin you create is just for your site. But many people like to share their plugins with the rest of the WordPress community. Before sharing your plugin, one thing you need to do is choose a license. This lets the user of your plugin know how they are allowed to use your code. To maintain compatibility with WordPress core, it is recommended that you pick a license that works with GNU General Public License (GPLv2+).

First published

September 16, 2014

Last updated

December 14, 2023

Previous What is a Plugin?

Next Header Requirements

Code is Poetry

Privacy

Are you writing a plugin that handles personal data – things like names, addresses, and other things that can be used to identify a person? You'll want to take care with that data and protect the privacy of your users and visitors.

What is Privacy?

WordPress.org made several enhancements ahead of Europe's General Data Protection Regulation. Following the launch of this work, we have made Privacy a permanent focus in core trac development, which will allow us to continue making enhancements on privacy and data protection outside specific legislation.

But what kind of issues might fall under the definition of "privacy", and how do we define it? Although privacy requirements vary widely across countries, cultures, and legal systems, there are several general principles applicable across any situation:

- **Consent and choice:** giving users (and site visitors) choices and options over the uses of their data, and requiring clear, specific, and informed opt-in;
- **Purpose legitimacy and specification:** only collect and use the personal data for the purpose it was intended for, and for which the user was clearly informed of in advance;
- **Collection limitation:** only collect the user data which is needed; don't make extra copies of data or combine your data with data from other plugins if you can avoid it
- **Data minimization:** restrict the processing of data, as well as the number of people who have access to it, to the minimum uses and people necessary;
- **Use, retention and disclosure limitation:** delete data which is no longer needed, both in active use and in archives, by both the recipient and any third parties;
- **Accuracy and quality:** ensure that the data collected and used is correct, relevant, and up-to-date, especially if inaccurate or poor data could adversely impact the user;
- **Openness, transparency and notice:** inform users how their data is being collected, used, and shared, as well as any rights they have over those uses;
- **Individual participation and access:** give users a means to access or download their data;
- **Accountability:** documenting the uses of data, protecting it in transit and in use by third parties, and preventing misuse and breaches as much as is possible;
- **Information security:** protecting data through appropriate technical and security measures;
- **Privacy compliance:** ensuring that the work meets the privacy regulations of the location where it will be used to collect and process people's data.

(Source: ISO 29100/Privacy Framework standard)

While not all of these principles will be applicable across all situations and uses, using them in the development

process can help to ensure user trust.

Privacy By Design

Many of these principles are espoused in the Privacy by Design framework, which states that:

- Privacy should be proactive, not reactive, and must anticipate privacy issues before they reach the user. Privacy must also be preventative, not remedial.
- Privacy should be the default setting. The user should not have to take actions to secure their privacy, and consent for data sharing should not be assumed.
- Privacy should be built into design as a core function, not an add-on.
- Privacy should be positive sum: there should be no trade-off between privacy and security, privacy and safety, or privacy and service provision.
- Privacy should offer end-to-end lifecycle protection through data minimization, minimal data retention, and regular deletion of data which is no longer required.
- The privacy standards used on your plugin (and service, if applicable) should be visible, transparent, open, documented and independently verifiable.
- Privacy should be user-centric. People should be given options such as granular privacy choices, maximized privacy defaults, detailed privacy information notices, user-friendly options, and clear notification of changes.

Food for Thought for Your Plugin

To help your plugin be ready, we recommend going through the following list of questions for every plugin that you make:

1. How does your plugin handle personal data? Use wp_add_privacy_policy_content (link) to disclose to your users any of the following:
    - Does the plugin share personal data with third parties (e.g. to outside APIs/servers). If so, what data does it share with which third parties and do they have a published privacy policy you can provide a link to?
    - Does the plugin collect personal data? If so, what data and where is it stored? Think about places like user data/meta, options, post meta, custom tables, files, etc.
    - Does the plugin use personal data collected by others? If so, what data? Does the plugin pass personal data to a SDK? What does that SDK do with the data?
    - Does the plugin collect telemetry data, directly or indirectly? Loading an image from a third-party source on every install, for example, could indirectly log and track the usage data of all of your plugin installs.
    - Does the plugin enqueue Javascript, tracking pixels or embed iframes from a third party (third party JS, tracking pixels and iframes can collect visitor's data/actions, leave cookies, etc.)?
    - Does the plugin store things in the browser? If so, where and what? Think about things like cookies, local storage, etc.
2. If your plugin collects personal data…
    - Does it provide a personal data exporter?
    - Does it provide a personal data eraser callback?
    - For what reasons (if any) does the plugin refuse to erase personal data? (e.g. order not yet completed, etc) – those should be disclosed as well.
3. Does the plugin use error logging? Does it avoid logging personal data if possible? Could you use things like wp_privacy_anonymize_data to minimize the personal data logged? How long are log entries kept? Who has access to them?
4. In wp-admin, what role/capabilities are required to access/see personal data? Are they sufficient?
5. What personal data is exposed on the front end of the site by the plugin? Does it appear to logged-in and logged-out users? Should it?
6. What personal data is exposed in REST API endpoints by the plugin? Does it appear to logged-in and logged-out users? What roles/capabilities are required to see it? Are those appropriate?
7. Does the plugin properly remove/clean-up data, including especially personal data:
    - During uninstall of the plugin?
    - When a related item is deleted (e.g. from the post meta or any post-referencing rows in another table)?
    - When a user is deleted (e.g. from any user referencing rows in a table)?

8. Does the plugin provide controls to reduce the amount of personal data required?
9. Does the plugin share personal data with SDKs or APIs only when the SDK or API requires it, or is the plugin also sharing personal data that is optional?
10. Does the amount of personal data collected or shared by this plugin change when certain other plugins are also installed?

External Resources

- Privacy Blog https://privacy.blog
- WordPress.org Privacy Policy https://wordpress.org/about/privacy/

First published

May 17, 2018

Last updated

December 14, 2023

Previous Advanced Topics

Next Adding the Personal Data Eraser to Your Plugin

Code is Poetry

Custom Post Types

WordPress stores the Post Types in the `posts` table allowing developers to register Custom Post Types along the ones that already exist.

This chapter will show you how to register Custom Post Types, how to retrieve their content from the database, and how to render them to the public.

First published

September 24, 2014

Last updated

December 14, 2023

Matt ↗
bbPress ↗
BuddyPress ↗

WordPress.org

Code is Poetry

Taxonomies

A **Taxonomy** is a fancy word for the classification/grouping of things. Taxonomies can be hierarchical (with parents/children) or flat.

WordPress stores the Taxonomies in the `term_taxonomy` database table allowing developers to register Custom Taxonomies along the ones that already exist.

Taxonomies have **Terms** which serve as the topic by which you classify/group things. They are stored inside the `terms` table.

For example: a Taxonomy named “Art” can have multiple Terms, such as “Modern” and “18th Century”.

This chapter will show you how to register Custom Taxonomies, how to retrieve their content from the database, and how to render them to the public.

> WordPress 3.4 and earlier had a Taxonomy named “Links” which was deprecated in WordPress 3.5.

First published

February 5, 2015

Last updated

December 14, 2023

Code is Poetry

Introduction to Plugin Development

Welcome to the Plugin Developer Handbook. Whether you're writing your first plugin or your fiftieth, we hope this resource helps you write the best plugin possible.

The Plugin Developer Handbook covers a variety of topics — everything from what should be in the plugin header, to security best practices, to tools you can use to build your plugin. It's also a work in progress — if you find something missing or incomplete, please notify the documentation team in slack and we'll make it better together.

Why We Make Plugins

If there's one cardinal rule in WordPress development, it's this: **Don't touch WordPress core**. This means that you don't edit core WordPress files to add functionality to your site. This is because WordPress overwrites core files with each update. Any functionality you want to add or modify should be done using plugins.

WordPress plugins can be as simple or as complicated as you need them to be, depending on what you want to do. The simplest plugin is a single PHP file. The Hello Dolly plugin is an example of such a plugin. The plugin PHP file just needs a Plugin Header, a couple of PHP functions, and some hooks to attach your functions to.

Plugins allow you to greatly extend the functionality of WordPress without touching WordPress core itself.

First published

July 30, 2014

Last updated

December 14, 2023

Code is Poetry

Header Requirements

As described in Getting Started, the main PHP file should include header comment what tells WordPress that a file is a plugin and provides information about the plugin.

Minimum Fields

At a minimum, a header comment must contain the Plugin Name:

```
/*
 * Plugin Name: YOUR PLUGIN NAME
 */
```

Header Fields

Available header fields:

- **Plugin Name:** (*required*) The name of your plugin, which will be displayed in the Plugins list in the WordPress Admin.
- **Plugin URI:** The home page of the plugin, which should be a unique URL, preferably on your own website. This *must be unique* to your plugin. You cannot use a WordPress.org URL here.
- **Description:** A short description of the plugin, as displayed in the Plugins section in the WordPress Admin. Keep this description to fewer than 140 characters.
- **Version:** The current version number of the plugin, such as 1.0 or 1.0.3.
- **Requires at least:** The lowest WordPress version that the plugin will work on.
- **Requires PHP:** The minimum required PHP version.
- **Author:** The name of the plugin author. Multiple authors may be listed using commas.
- **Author URI:** The author's website or profile on another website, such as WordPress.org.
- **License:** The short name (slug) of the plugin's license (e.g. GPLv2). More information about licensing can be found in the WordPress.org guidelines.
- **License URI:** A link to the full text of the license (e.g. https://www.gnu.org/licenses/gpl-2.0.html).
- **Text Domain:** The gettext text domain of the plugin. More information can be found in the Text Domain section of the How to Internationalize your Plugin page.
- **Domain Path:** The domain path lets WordPress know where to find the translations. More information can be found in the Domain Path section of the How to Internationalize your Plugin page.
- **Network:** Whether the plugin can only be activated network-wide. Can only be set to *true*, and should be left out when not needed.
- **Update URI:** Allows third-party plugins to avoid accidentally being overwritten with an update of a plugin of a

similar name from the WordPress.org Plugin Directory. For more info read related dev note.
- **Requires Plugins**: A comma-separated list of WordPress.org-formatted slugs for its dependencies, such as `my-plugin` (`my-plugin/my-plugin.php` is not supported). It does not support commas in plugin slugs. For more info read the related dev note.

A valid PHP file with a header comment might look like this:

```
/*
 * Plugin Name:       My Basics Plugin
 * Plugin URI:        https://example.com/plugins/the-basics/
 * Description:       Handle the basics with this plugin.
 * Version:           1.10.3
 * Requires at least: 5.2
 * Requires PHP:      7.2
 * Author:            John Smith
 * Author URI:        https://author.example.com/
 * License:           GPL v2 or later
 * License URI:       https://www.gnu.org/licenses/gpl-2.0.html
 * Update URI:        https://example.com/my-plugin/
 * Text Domain:       my-basics-plugin
 * Domain Path:       /languages
 * Requires Plugins:  my-plugin, yet-another-plugin
 */
```

Here's another example which allows file-level PHPDoc DocBlock as well as WordPress plugin file headers:

```
/**
 * Plugin Name
 *
 * @package           PluginPackage
 * @author            Your Name
 * @copyright         2019 Your Name or Company Name
 * @license           GPL-2.0-or-later
 *
 * @wordpress-plugin
 * Plugin Name:       Plugin Name
 * Plugin URI:        https://example.com/plugin-name
 * Description:       Description of the plugin.
 * Version:           1.0.0
 * Requires at least: 5.2
 * Requires PHP:      7.2
 * Author:            Your Name
 * Author URI:        https://example.com
 * Text Domain:       plugin-slug
 * License:           GPL v2 or later
 * License URI:       http://www.gnu.org/licenses/gpl-2.0.txt
 * Update URI:        https://example.com/my-plugin/
 * Requires Plugins:  my-plugin, yet-another-plugin
 */
```

Notes

When assigning a version number to your project, keep in mind that WordPress uses the PHP version_compare() function to compare plugin version numbers. Therefore, before you release a new version of your plugin, you should make sure that this PHP function considers the new version to be “greater” than the old one. For example, 1.02 is actually greater than 1.1.

First published

September 16, 2014

Last updated

May 8, 2024

Previous Plugin Basics

Next Activation / Deactivation Hooks

Activation / Deactivation Hooks

Activation and deactivation hooks provide ways to perform actions when plugins are activated or deactivated.

- On *activation*, plugins can run a routine to add rewrite rules, add custom database tables, or set default option values.
- On *deactivation*, plugins can run a routine to remove temporary data such as cache and temp files and directories.

> The deactivation hook is sometimes confused with the uninstall hook. The uninstall hook is best suited to **delete all data permanently** such as deleting plugin options and custom tables, etc.

Activation

To set up an activation hook, use the register_activation_hook() function:

```
register_activation_hook(
    __FILE__,
    'pluginprefix_function_to_run'
);
```

Deactivation

To set up a deactivation hook, use the register_deactivation_hook() function:

```
register_deactivation_hook(
    __FILE__,
    'pluginprefix_function_to_run'
);
```

The first parameter in each of these functions refers to your main plugin file, which is the file in which you have placed the plugin header comment. Usually these two functions will be triggered from within the main plugin file; however, if the functions are placed in any other file, you must update the first parameter to correctly point to the main plugin file.

Example

One of the most common uses for an activation hook is to refresh WordPress permalinks when a plugin registers a custom post type. This gets rid of the nasty 404 errors.

Let's look at an example of how to do this:

```
/**
 * Register the "book" custom post type
 */
function pluginprefix_setup_post_type() {
    register_post_type( 'book', ['public' => true ] );
}
add_action( 'init', 'pluginprefix_setup_post_type' );


/**
 * Activate the plugin.
 */
function pluginprefix_activate() {
    // Trigger our function that registers the custom post type plugin.
    pluginprefix_setup_post_type();
    // Clear the permalinks after the post type has been registered.
    flush_rewrite_rules();
}
register_activation_hook( __FILE__, 'pluginprefix_activate' );
```

If you are unfamiliar with registering custom post types, don't worry – this will be covered later. This example is used simply because it's very common.

Using the example from above, the following is how to reverse this process and deactivate a plugin:

```
/**
 * Deactivation hook.
 */
function pluginprefix_deactivate() {
    // Unregister the post type, so the rules are no longer in memory.
    unregister_post_type( 'book' );
    // Clear the permalinks to remove our post type's rules from the database.
    flush_rewrite_rules();
}
register_deactivation_hook( __FILE__, 'pluginprefix_deactivate' );
```

For further information regarding activation and deactivation hooks, here are some excellent resources:

- register_activation_hook() in the WordPress function reference.
- register_deactivation_hook() in the WordPress function reference.

First published

September 16, 2014

Last updated

February 20, 2024

Code is Poetry

Uninstall Methods

Your plugin may need to do some clean-up when it is uninstalled from a site.

A plugin is considered uninstalled if a user has deactivated the plugin, and then clicks the delete link within the WordPress Admin.

When your plugin is uninstalled, you'll want to clear out any plugin options and/or settings specific to the plugin, and/or other database entities such as tables.

Less experienced developers sometimes make the mistake of using the deactivation hook for this purpose.

This table illustrates the differences between deactivation and uninstall.

Scenario	Deactivation Hook	Uninstall Hook
Flush Cache/Temp	Yes	No
Flush Permalinks	Yes	No
Remove Options from {$wpdb->prefix}_options	No	Yes
Remove Tables from wpdb	No	Yes

Method 1: register_uninstall_hook

To set up an uninstall hook, use the register_uninstall_hook() function:

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

Method 2: uninstall.php

To use this method you need to create an `uninstall.php` file inside the root folder of your plugin. This magic file is

run automatically when the users deletes the plugin.

For example: `/plugin-name/uninstall.php`

> Always check for the constant `WP_UNINSTALL_PLUGIN` in `uninstall.php` before doing anything. This protects against direct access.
>
> The constant will be defined by WordPress during the `uninstall.php` invocation.
>
> The constant is **NOT** defined when uninstall is performed by register_uninstall_hook() .

Here is an example deleting option entries and dropping a database table:

```
// if uninstall.php is not called by WordPress, die
if ( ! defined( 'WP_UNINSTALL_PLUGIN' ) ) {
    die;
}

$option_name = 'wporg_option';

delete_option( $option_name );

// for site options in Multisite
delete_site_option( $option_name );

// drop a custom database table
global $wpdb;
$wpdb->query( "DROP TABLE IF EXISTS {$wpdb->prefix}mytable" );
```

> In Multisite, looping through all blogs to delete options can be very resource intensive.

First published

September 16, 2014

Last updated

February 20, 2024

Previous Including a Software License

Next Plugin Security

About
News
Hosting
Privacy

Showcase
Themes

Code is Poetry

What is a Plugin?

Plugins are packages of code that extend the core functionality of WordPress. WordPress plugins are made up of PHP code and can include other assets such as images, CSS, and JavaScript.

By making your own plugin you are *extending* WordPress, i.e. building additional functionality on top of what WordPress already offers. For example, you could write a plugin that displays links to the ten most recent posts on your site.

Or, using WordPress' custom post types, you could write a plugin that creates a full-featured support ticketing system with email notifications, custom ticket statuses, and a client-facing portal. The possibilities are *endless!*

Most WordPress plugins are composed of many files, but a plugin really only *needs* one main file with a specifically formatted DocBlock in the header.

Hello Dolly, one of the first plugins, is only 100 lines long. Hello Dolly shows lyrics from the famous song in the WordPress admin. Some CSS is used in the PHP file to control how the lyric is styled.

As a WordPress.org plugin author, you have an amazing opportunity to create a plugin that will be installed, tinkered with, and loved by millions of WordPress users. All **you** need to do is turn your great idea into code. The Plugin Handbook is here to help you with that.

First published

July 30, 2014

Last updated

May 2, 2021

Code is Poetry

Advanced Topics

Removing Actions and Filters

Sometimes you want to remove a callback function from a hook that another plugin, theme or even WordPress Core has registered.

To remove a callback function from a hook, you need to call `remove_action()` or `remove_filter()`, depending whether the callback function was added as an Action or a Filter.

The parameters passed to `remove_action()` / `remove_filter()` must be identical to the parameters passed to `add_action()` / `add_filter()` that registered it, or the removal won't work.

> To successfully remove a callback function you must perform the removal after the callback function was registered. The order of execution is important.

Example

Lets say we want to improve the performance of a large theme by removing unnecessary functionality.

Let's analyze the theme's code by looking into `functions.php`.

```
function wporg_setup_slider() {
    // ...
}
add_action( 'template_redirect', 'wporg_setup_slider', 9 );
```

The `wporg_setup_slider` function is adding a slider that we don't need, which probably loads a huge CSS file followed by a JavaScript initialization file which uses a custom written library the size of 1MB. We can can get rid of that.

Since we want to hook into WordPress after the `wporg_setup_slider` callback function was registered (`functions.php` executed) our best chance would be the `after_setup_theme` hook.

```
function wporg_disable_slider() {
```

```
    // Make sure all parameters match the add_action() call exactly.
    remove_action( 'template_redirect', 'wporg_setup_slider', 9 );
}
// Make sure we call remove_action() after add_action() has been called.
add_action( 'after_setup_theme', 'wporg_disable_slider' );
```

Removing All Callbacks

You can also remove all of the callback functions associated with a hook by using `remove_all_actions()` / `remove_all_filters()`.

Determining the Current Hook

Sometimes you want to run an Action or a Filter on multiple hooks, but behave differently based on which one is currently calling it.

You can use the `current_action()` / `current_filter()` to determine the current Action / Filter.

```
function wporg_modify_content( $content ) {
    switch ( current_filter() ) {
        case 'the_content':
            // Do something.
            break;
        case 'the_excerpt':
            // Do something.
            break;
    }
    return $content;
}

add_filter( 'the_content', 'wporg_modify_content' );
add_filter( 'the_excerpt', 'wporg_modify_content' );
```

Checking How Many Times a Hook Has Run

Some hooks are called multiple times in the course of execution, but you may only want your callback function to run once.

In this situation, you can check how many times the hook has run with the did_action().

```
function wporg_custom() {
   // If save_post has been run more than once, skip the rest of the code.
   if ( did_action( 'save_post' ) !== 1 ) {
      return;
   }
   // ...
}
add_action( 'save_post', 'wporg_custom' );
```

Debugging with the “all” Hook

If you want a callback function to fire on every single hook, you can register it to the `all` hook. Sometimes this is useful in debugging situations to help determine when a particular event is happening or when a page is crashing.

```
function wporg_debug() {
```

```
    echo '<p>' . current_action() . '</p>';
}
add_action( 'all', 'wporg_debug' );
```

First published

September 16, 2014

Last updated

March 20, 2025

Previous Custom Hooks

Next Privacy

Adding the Personal Data Eraser to Your Plugin

In WordPress 4.9.6, new tools were added to make compliance easier with laws like the European Union's General Data Protection Regulation, or GDPR for short. Among the tools added is a Personal Data Removal tool which supports erasing/anonymizing personal data for a given user. It does NOT delete registered user accounts – that is still a separate step the admin can choose whether or not to do.

In addition to the personal data stored in things like WordPress comments, plugins can also hook into the eraser feature to erase the personal data they collect, whether it be in something like postmeta or even an entirely new Custom Post Type (CPT).

Like the exporters, the "key" for all the erasers is the user's email address – this was chosen because it supports erasing personal data for both full-fledged registered users and also unregistered users (e.g. like a logged out commenter).

However, since performing a personal data erase is a destructive process, we don't want to just do it without confirming the request, so the admin-facing user interface starts all requests by having the admin enter the username or email address making the request and then sends then a link to click to confirm their request. Once a request has been confirmed, the admin can kick off personal data erasure for the user, or force one if the need arises.

The way the personal data export is erased is similar to how the personal data exporters – and relies on hooking "eraser" callbacks to do the dirty work of erasing the data. When the admin clicks on the remove personal data link, an AJAX loop begins that iterates over all the erasers registered in the system, one at a time. In addition to erasers built into core, plugins can register their own eraser callbacks.

The eraser callback interface is designed to be as simple as possible. An eraser callback receives the email address we are working with, and a page parameter as well. The page parameter (which starts at 1) is used to avoid plugins potentially causing timeouts by attempting to erase all the personal data they've collected at once. A well behaved plugin will limit the amount of data it attempts to erase per page (e.g. 100 posts, 200 comments, etc.)

The eraser callback replies whether items containing personal data were erased, whether any items containing personal data were retained, an array of messages to present to the admin (explaining why items that were retained were retained) and whether it is done or not. If an eraser callback reports that it is not done, it will be called again (in a separate request) with the page parameter incremented by 1.

When all the exporters have been called to completion, the admin user interface is updated to show whether or not all personal data found was erased, and any messages explaining why personal data was retained.

Let's work on a hypothetical plugin which adds commenter location data to comments. Let's assume the plugin has used `add_comment_meta` to add location data using `meta_keys` of `latitude` and `longitude`

The first thing the plugin needs to do is to create an eraser function that accepts an email address and a page, e.g.:

```
/**
 * Removes any stored location data from a user's comment meta for the supplied email address.
 *
 * @param string $email_address   email address to manipulate
 * @param int    $page            pagination
 *
 * @return array
 */
function wporg_remove_location_meta_from_comments_for_email( $email_address, $page = 1 ) {
    $number = 500; // Limit us to avoid timing out
    $page   = (int) $page;

    $comments = get_comments(
        array(
            'author_email' => $email_address,
            'number'       => $number,
            'paged'        => $page,
            'order_by'     => 'comment_ID',
            'order'        => 'ASC',
        )
    );

    $items_removed = false;

    foreach ( (array) $comments as $comment ) {
        $latitude  = get_comment_meta( $comment->comment_ID, 'latitude', true );
        $longitude = get_comment_meta( $comment->comment_ID, 'longitude', true );

        if ( ! empty( $latitude ) ) {
            delete_comment_meta( $comment->comment_ID, 'latitude' );
            $items_removed = true;
        }

        if ( ! empty( $longitude ) ) {
            delete_comment_meta( $comment->comment_ID, 'longitude' );
            $items_removed = true;
        }
    }

    // Tell core if we have more comments to work on still
    $done = count( $comments ) < $number;
    return array(
        'items_removed'  => $items_removed,
        'items_retained' => false, // always false in this example
        'messages'       => array(), // no messages in this example
        'done'           => $done,
    );
}
```

The next thing the plugin needs to do is to register the callback by filtering the eraser array using the
`wp_privacy_personal_data_erasers`
filter.

When registering you provide a friendly name for the eraser (to aid in debugging – this friendly name is not shown to anyone at this time) and the callback, e.g.

```
/**
 * Registers all data erasers.
 *
 * @param array $exporters
 *
 * @return mixed
 */
function wporg_register_privacy_erasers( $erasers ) {
    $erasers['my-plugin-slug'] = array(
        'eraser_friendly_name' => __( 'Comment Location Plugin', 'text-domain' ),
        'callback'             => 'wporg_remove_location_meta_from_comments_for_email',
    );
    return $erasers;
}

add_filter( 'wp_privacy_personal_data_erasers', 'wporg_register_privacy_erasers' );
```

And that’s all there is to it! Your plugin will now clean up its personal data!

First published

May 17, 2018

Last updated

September 16, 2024

Previous Privacy

Next Adding the Personal Data Exporter to Your Plugin

WordPress.org

Code is Poetry

Registering Custom Post Types

WordPress comes with five default post types: `post`, `page`, `attachment`, `revision`, and `menu`.

While developing your plugin, you may need to create your own specific content type: for example, products for an e-commerce website, assignments for an e-learning website, or movies for a review website.

Using Custom Post Types, you can register your own post type. Once a custom post type is registered, it gets a new top-level administrative screen that can be used to manage and create posts of that type.

To register a new post type, you use the register_post_type() function.

> We recommend that you put custom post types in a plugin rather than a theme. This ensures that user content remains portable even if the theme is changed.

The following minimal example registers a new post type, Products, which is identified in the database as `wporg_product`.

```
function wporg_custom_post_type() {
    register_post_type('wporg_product',
        array(
            'labels'      => array(
                'name'          => __('Products', 'textdomain'),
                'singular_name' => __('Product', 'textdomain'),
            ),
                'public'      => true,
                'has_archive' => true,
        )
    );
}
add_action('init', 'wporg_custom_post_type');
```

Please visit the reference page for register_post_type() for the description of arguments.

> You must call `register_post_type()` before the `admin_init` hook and after the `after_setup_theme` hook. A good hook to use is the `init` action hook.

Naming Best Practices

It is important that you prefix your post type functions and identifiers with a short prefix that corresponds to your plugin, theme, or website.

> **Make sure your custom post type identifier does not exceed 20 characters** as the `post_type` column in the database is currently a VARCHAR field of that length.

> **To ensure forward compatibility**, do not use **wp_** as your identifier — it is being used by WordPress core.

> If your identifier is too generic (for example: "`product`"), it may conflict with other plugins or themes that have chosen to use that same identifier.

> **Solving duplicate post type identifiers is not possible without disabling one of the conflicting post types.**

URLs

A custom post type gets its own slug within the site URL structure.

A post of type `wporg_product` will use the following URL structure by default: `http://example.com/wporg_product/%product_name%`.

`wporg_product` is the slug of your custom post type and `%product_name%` is the slug of your particular product.

The final permalink would be: `http://example.com/wporg_product/wporg-is-awesome`.

You can see the permalink on the edit screen for your custom post type, just like with default post types.

A Custom Slug for a Custom Post Type

To set a custom slug for the slug of your custom post type all you need to do is add a key => value pair to the `rewrite` key in the `register_post_type()` arguments array.

Example:

```
function wporg_custom_post_type() {
    register_post_type('wporg_product',
        array(
            'labels'      => array(
                'name'          => __( 'Products', 'textdomain' ),
                'singular_name' => __( 'Product', 'textdomain' ),
```

```
            ),
            'public'      => true,
            'has_archive' => true,
            'rewrite'     => array( 'slug' => 'products' ), // my custom slug
        )
    );
}
add_action('init', 'wporg_custom_post_type');
```

The above will result in the following URL structure: `http://example.com/products/%product_name%`

Using a generic slug like `products` can potentially conflict with other plugins or themes, so try to use one that is more specific to your content.

Unlike the custom post type identifiers, the duplicate slug problem can be solved easily by changing the slug for one of the conflicting post types.

If the plugin author included an `apply_filters()` call on the arguments, this can be done programmatically by overriding the arguments submitted via the `register_post_type()` function.

First published

September 24, 2014

Last updated

November 17, 2022

Previous Custom Post Types

Next Working with Custom Post Types

WordPress.com ↗
Matt ↗
bbPress ↗
BuddyPress ↗

WordPress.org

Code is Poetry

Working with Custom Post Types

Custom Post Type Templates

You can create custom templates for your custom post types. In the same way posts and their archives can be displayed using `single.php` and `archive.php`, you can create the templates:

- `single-{post_type}.php` – for single posts of a custom post type
- `archive-{post_type}.php` – for the archive

Where `{post_type}` is the post type identifier, used as the `$post_type` argument of the `register_post_type()` function.

Building upon what we've learned previously, you could create `single-wporg_product.php` and `archive-wporg_product.php` template files for single product posts and the archive.

Alternatively, you can use the is_post_type_archive() function in any template file to check if the query shows an archive page of a given post type, and the post_type_archive_title() function to display the post type title.

Querying by Post Type

You can query posts of a specific type by passing the `post_type` key in the arguments array of the `WP_Query` class constructor.

```
<?php
$args = array(
    'post_type'      => 'product',
    'posts_per_page' => 10,
);
$loop = new WP_Query($args);
while ( $loop->have_posts() ) {
    $loop->the_post();
    ?>
    <div class="entry-content">
        <?php the_title(); ?>
        <?php the_content(); ?>
    </div>
    <?php
```

```
}
```

This loops through the latest ten product posts and displays the title and content of them one by one.

Altering the Main Query

Registering a custom post type does not mean it gets added to the main query automatically.

If you want your custom post type posts to show up on standard archives or include them on your home page mixed up with other post types, use the `pre_get_posts` action hook.

The next example will show posts from `post`, `page` and `movie` post types on the home page:

```
function wporg_add_custom_post_types($query) {
    if ( is_home() && $query->is_main_query() ) {
        $query->set( 'post_type', array( 'post', 'page', 'movie' ) );
    }
    return $query;
}
add_action('pre_get_posts', 'wporg_add_custom_post_types');
```

First published

September 24, 2014

Last updated

November 17, 2022

Previous Registering Custom Post Types

Next Taxonomies

WordPress.org

Code is Poetry

Rendering Post Metadata

Here is a non exhaustive list of functions and template tags used to get and render Post Metadata:

- the_meta() – Template tag that automatically lists all Custom Fields of a post
- get_post_custom() and get_post_meta() – Retrieves one or all metadata of a post.
- get_post_custom_values() – Retrieves values for a custom post field.

First published

September 24, 2014

Last updated

February 8, 2024

Matt ↗
bbPress ↗
BuddyPress ↗

WordPress.org

Code is Poetry

Term Splitting (WordPress 4.2)

This information is here for historical purposes. If you're not interested in how terms worked prior to 2015, you can skip this section.

Prior to WordPress 4.2

Terms in different taxonomies with the same slug shared a single term ID. For instance, a tag and a category with the slug "news" had the same term ID.

WordPress 4.2+

Beginning with 4.2, when one of these shared terms is updated, it will be split: the updated term will be assigned a new term ID.

What does it mean for you?

In the vast majority of situations, this update was seamless and uneventful. However, some plugins and themes who store term IDs in options, post meta, user meta, or elsewhere might have been affected.

Handling the Split

WordPress 4.2 includes two different tools to help authors of plugins and themes with the transition.

The split_shared_term hook

When a shared term is assigned a new term ID, a new `split_shared_term` action is fired.

Here are a few examples of how plugin and theme authors can leverage this hook to ensure that stored term IDs are updated.

Term ID stored in an option

Let's say your plugin stores an option called `featured_tags` that contains an array of term IDs (`[4, 6, 10]`) that serve as the query parameter for your homepage featured posts section.

In this example, you'll hook to `split_shared_term` action, check whether the updated term ID is in the array, and

update if necessary.

```
/**
 * Update featured_tags option when a shared term gets split.
 *
 * @param int    $term_id          ID of the formerly shared term.
 * @param int    $new_term_id      ID of the new term created for the $term_taxonomy_id.
 * @param int    $term_taxonomy_id ID for the term_taxonomy row affected by the split.
 * @param string $taxonomy         Taxonomy for the split term.
 */
function wporg_featured_tags_split( int $term_id, int $new_term_id, int $term_taxonomy_id, string
$taxonomy ): void {
    // we only care about tags, so we'll first verify that the taxonomy is post_tag.
    if ( 'post_tag' === $taxonomy ) {

        // get the currently featured tags.
        $featured_tags = get_option( 'featured_tags' );

        // if the updated term is in the array, note the array key.
        $found_term = array_search( $term_id, $featured_tags, true );
        if ( false !== $found_term ) {

            // the updated term is a featured tag! replace it in the array, save the new array.
            $featured_tags[ $found_term ] = $new_term_id;
            update_option( 'featured_tags', $featured_tags );
        }
    }
}
add_action( 'split_shared_term', 'wporg_featured_tags_split', 10, 4 );
```

Term ID stored in post meta

Let’s say your plugin stores a term ID in post meta for pages so that you can show related posts for a certain page.

In this case, you need to use the `get_posts()` function to get the pages with your `meta_key` and update the `meta_value` matching the split term ID.

```
/**
 * Update related posts term ID for pages
 *
 * @param int    $term_id          ID of the formerly shared term.
 * @param int    $new_term_id      ID of the new term created for the $term_taxonomy_id.
 * @param int    $term_taxonomy_id ID for the term_taxonomy row affected by the split.
 * @param string $taxonomy         Taxonomy for the split term.
 */
function wporg_page_related_posts_split( int $term_id, int $new_term_id, int $term_taxonomy_id,
string $taxonomy ): void {
    // find all the pages where meta_value matches the old term ID.
    $page_ids = get_posts(
        array(
            'post_type'  => 'page',
            'fields'     => 'ids',
            'meta_key'   => 'meta_key',
            'meta_value' => $term_id,
        )
```

```
    );

    // if such pages exist, update the term ID for each page.
    if ( $page_ids ) {
        foreach ( $page_ids as $id ) {
            update_post_meta( $id, 'meta_key', $new_term_id, $term_id );
        }
    }
}
add_action( 'split_shared_term', 'wporg_page_related_posts_split', 10, 4 );
```

The wp_get_split_term function

> Using the `split_shared_term` hook is the preferred method for processing Term ID changes.
>
> However, there may be cases where Terms are split without your plugin having a chance to hook to the `split_shared_term` action.

WordPress 4.2 stores information about taxonomy terms that have been split, and provides the `wp_get_split_term()` utility function to help developers retrieve this information.

Consider the case above, where your plugin stores term IDs in an option named `featured_tags`. You may want to build a function that validates these tag IDs (perhaps to be run on plugin update), to be sure that none of the featured tags has been split:

```
/**
 * Retrieve information about split terms and udpates the featured_tags option with the new term
IDs.
 *
 * @return void
 */
function wporg_featured_tags_check_split() {
    $featured_tag_ids = get_option( 'featured_tags', array() );

    // check to see whether any IDs correspond to post_tag terms that have been split.
    foreach ( $featured_tag_ids as $index => $featured_tag_id ) {
        $new_term_id = wp_get_split_term( $featured_tag_id, 'post_tag' );

        if ( $new_term_id ) {
            $featured_tag_ids[ $index ] = $new_term_id;
        }
    }

    // save
    update_option( 'featured_tags', $featured_tag_ids );
}
```

Note that `wp_get_split_term()` takes two parameters, `$old_term_id` and `$taxonomy` and returns an integer.

If you need to retrieve a list of all split terms associated with an old Term ID, regardless of taxonomy, use `wp_get_split_terms()`.

First published

February 5, 2015

Last updated

June 19, 2023

Previous Taxonomies

Next Working with Custom Taxonomies

Code is Poetry

Hooks

Hooks are a way for one piece of code to interact/modify another piece of code at specific, pre-defined spots. They make up the foundation for how plugins and themes interact with WordPress Core, but they're also used extensively by Core itself.

There are two types of hooks: Actions and Filters. To use either, you need to write a custom function known as a `Callback`, and then register it with a WordPress hook for a specific action or filter.

Actions allow you to add data or change how WordPress operates. Actions will run at a specific point in the execution of WordPress Core, plugins, and themes. Callback functions for Actions can perform some kind of a task, like echoing output to the user or inserting something into the database. Callback functions for an Action do not return anything back to the calling Action hook.

Filters give you the ability to change data during the execution of WordPress Core, plugins, and themes. Callback functions for Filters will accept a variable, modify it, and return it. They are meant to work in an isolated manner, and should never have side effects such as affecting global variables and output. Filters expect to have something returned back to them.

WordPress provides many hooks that you can use, but you can also create your own so that other developers can extend and modify your plugin or theme.

Actions vs. Filters

The main difference between an action and a filter can be summed up like this:

- an action takes the info it receives, does something with it, and returns nothing. In other words: it *acts* on something and then exits, returning nothing back to the calling hook.
- a filter takes the info it receives, modifies it somehow, and returns it. In other words: it *filters* something and passes it back to the hook for further use.

Said another way:

- an action interrupts the code flow to do something, and then returns back to the normal flow without modifying anything;
- a filter is used to modify something in a specific way so that the modification is then used by code later on.

The *something* referred to is the parameter list sent via the hook definition. More on this in later sections.

More Resources

- Filter Reference
- Action Reference

First published

September 16, 2014

Last updated

January 29, 2024

Previous Securing (sanitizing) Input

Next Actions