Plugin Basics

- Plugin Basics
- Activation / Deactivation Hooks
- Best Practices
- Determining Plugin and Content Directories
- Header Requirements
- Including a Software License
- Uninstall Methods

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:

```
$ cd wp-content/
$ cd plugins/
$ mkdir plugin-name/
$ cd plugin-name/
$ vim plugin-name.php
```

In the example above, `vim` is the name of the text editor. Use whichever editor 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+).

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.

[alert]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.[/alert]

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.

Best Practices

Here are some best practices to help organize your code so it works well alongside WordPress core and other WordPress plugins.

Avoid Naming Collisions

A naming collision happens when your plugin is using the same name for a variable, function or a class as another plugin.

Luckily, you can avoid naming collisions by using the methods below.

Procedural Coding Method

By default, all variables, functions and classes are defined in the **global namespace**, which means that it is possible for your plugin to override variables, functions and classes set by another plugin and vice-versa. Variables that are defined *inside* of functions or classes are not affected by this.

Prefix Everything

All globally accessible code should be prefixed with a *unique* identifier. Prefixes prevent conflicts with other plugins and prevents them from overwriting your variables and accidentally calling your functions and classes.

In order to prevent conflicts with other plugins, your prefix should be at least 4 letters long, though we recommend 5. You should avoid using a common English word, and instead choose something unique to your plugin. We host tens of thousands of plugins on WordPress.org alone. There are hundreds of thousands more outside our servers. You're *going* to run into conflicts.

A good way to do this is with a prefix. For example, if your plugin is called "Easy Custom Post Types" then you could use names like these:

- `function ecpt_save_post()`
- `define( 'ECPT_LICENSE', true );`
- `class ECPT_Admin{}`
- `namespace EasyCustomPostTypes;`
- `update_option( 'ecpt_settings', $settings );`

[info]Because you are making code as a part of the **WordPress** project, you must avoid the use of prefixes that have a high probability of conflicting with the core WordPress. This includes but is not limited to: `__` (double underscores), `wp_` , `WordPress`, or `_` (single underscore)

If you are making code for a 'sub' plugin (such as a WooCommece extension), you would similarly need to avoid using any of their normal/common prefixes (i.e. Woo, WooCommerce).

You can use them *inside* your classes or namespace, but not as stand-alone function/namespace/class.[/info]

If you're using `_n()` or `__()` for translation, that's fine. We're **only** talking about functions you've created for your plugin, not the core functions from WordPress. In fact, those core features are *why* you need to not use those prefixes in your own plugin! You wouldn't want to break WordPress for your users.

Remember: Good prefix names are unique and distinct to your plugin. This will help you and the next person in debugging, as well as prevent conflicts.

Code that **must** be prefixed includes:

- Functions (unless namespaced)
- Classes, interfaces, and traits (unless namespaced)
- Namespaces
- Global variables
- Options and transients

Check for Existing Implementations

PHP provides a number of functions to verify existence of variables, functions, classes and constants. All of these will return true if the entity exists.

- **Variables**: isset() (includes arrays, objects, etc.)
- **Functions**: function_exists()
- **Classes**: class_exists()
- **Constants**: defined()

Keep in mind that using `(!function_exists('NAME')) {` around all your functions and classes sounds like a great idea until you realize the fatal flaw. If something else has a function with the same name and their code loads first, your plugin will break. Using if-exists to replace/override a function or class should be reserved for *shared* libraries only.

Example

```
// Create a function called "wporg_init" if it doesn't already exist
if ( ! function_exists( 'wporg_init' ) ) {
  function wporg_init() {
    register_setting( 'wporg_settings', 'wporg_option_foo' );
  }
}

// Create a function called "wporg_get_foo" if it doesn't already exist
if ( ! function_exists( 'wporg_get_foo' ) ) {
  function wporg_get_foo() {
    return get_option( 'wporg_option_foo' );
  }
}
```

Object Oriented Programming Method

An easier way to tackle the naming collision problem is to use a class for the code of your plugin.

You will still need to take care of checking whether the name of the class you want is already taken but the rest will be taken care of by PHP.

Example

```
if ( ! class_exists( 'WPOrg_Plugin' ) ) {
  class WPOrg_Plugin {
    public static function init() {
      register_setting( 'wporg_settings', 'wporg_option_foo' );
    }

    public static function get_foo() {
      return get_option( 'wporg_option_foo' );
    }
  }
```

```
  WPOrg_Plugin::init();
  WPOrg_Plugin::get_foo();
}
```

File Organization

The root level of your plugin directory should contain your `plugin-name.php` file and, optionally, your uninstall.php file. All other files should be organized into sub folders whenever possible.

Folder Structure

A clear folder structure helps you and others working on your plugin keep similar files together.

Here's a sample folder structure for reference:

```
/plugin-name
  plugin-name.php
  uninstall.php
  /languages
  /includes
  /admin
    /js
    /css
    /images
  /public
    /js
    /css
    /images
```

Plugin Architecture

The architecture, or code organization, you choose for your plugin will likely depend on the size of your plugin.

For small, single-purpose plugins that have limited interaction with WordPress core, themes or other plugins, there's little benefit in engineering complex classes; unless you know the plugin is going to expand greatly later on.

For large plugins with lots of code, start off with classes in mind. Separate style and scripts files, and even build-related files. This will help code organization and long-term maintenance of the plugin.

Conditional Loading

It's helpful to separate your admin code from the public code. Use the conditional is_admin(). You must still perform capability checks as this doesn't indicate the user is authenticated or has Administrator-level access. See Checking User Capabilities.

For example:

```
if ( is_admin() ) {
  // we are in admin mode
  require_once __DIR__ . '/admin/plugin-name-admin.php';
}
```

Avoiding Direct File Access

As a security precaution, it's a good practice to disallow access if the `ABSPATH` global is not defined. This is only applicable to files which contain code outside of class or function definitions, such as the main plugin file.

You can implement this by including this code at the top of the file:

```
if ( ! defined( 'ABSPATH' ) ) {
  exit; // Exit if accessed directly
}
```

Architecture Patterns

While there are a number of possible architecture patterns, they can broadly be grouped into three variations:

- Single plugin file, containing functions
- Single plugin file, containing a class, instantiated object and optionally functions
- Main plugin file, then one or more class files

Architecture Patterns Explained

Specific implementations of the more complex of the above code organizations have already been written up as tutorials and slides:

- Slash – Singletons, Loaders, Actions, Screens, Handlers
- Implementing the MVC Pattern in WordPress Plugins

Boilerplate Starting Points

Instead of starting from scratch for each new plugin you write, you may want to start with a **boilerplate**. One advantage of using a boilerplate is to have consistency among your own plugins. Boilerplates also make it easier for other people to contribute to your code if you use a boilerplate they are already familiar with.

These also serve as further examples of different yet comparable architectures.

- WordPress Plugin Boilerplate: A foundation for WordPress Plugin Development that aims to provide a clear and consistent guide for building your plugins.
- WordPress Plugin Bootstrap: Basic bootstrap to develop WordPress plugins using Grunt, Compass, GIT, and SVN.
- WP Skeleton Plugin: Skeleton plugin that focuses on unit tests and use of composer for development.
- WP CLI Scaffold: The Scaffold command of WP CLI creates a skeleton plugin with options such as CI configuration files.

Of course, you could take different aspects of these and others to create your own custom boilerplate.

Determining Plugin and Content Directories

When coding WordPress plugins you often need to reference various files and folders throughout the WordPress installation and within your plugin or theme.

WordPress provides several functions for easily determining where a given file or directory lives. Always use these functions in your plugins instead of hard-coding references to the wp-content directory or using the WordPress internal constants.

[tip]WordPress allows users to place their wp-content directory anywhere they want and rename it whatever they want. Never assume that plugins will be in wp-content/plugins, uploads will be in wp-content/uploads, or that themes will be in wp-content/themes.[/tip]

PHP's `__FILE__` magic-constant resolves symlinks automatically, so if the `wp-content/` or `wp-content/plugins/` or even the individual plugin directory is symlinked, hardcoded paths will not work correctly.

Common Usage

If your plugin includes JavaScript files, CSS files or other external files, then it's likely you'll need the URL to these files so you can load them into the page. To do this you should use the plugins_url() function like so:

```
plugins_url( 'myscript.js', __FILE__ );
```

This will return the full URL to myscript.js, such as `example.com/wp-content/plugins/myplugin/myscript.js`.

To load your plugins' JavaScript or CSS into the page you should use `wp_enqueue_script()` or `wp_enqueue_style()` respectively, passing the result of `plugins_url()` as the file URL.

Available Functions

WordPress includes many other functions for determining paths and URLs to files or directories within plugins, themes, and WordPress itself. See the individual DevHub pages for each function for complete information on their use.

Plugins

- `plugins_url()`
- `plugin_dir_url()`
- `plugin_dir_path()`
- `plugin_basename()`

Themes

- `get_template_directory_uri()`
- `get_stylesheet_directory_uri()`
- `get_stylesheet_uri()`
- `get_theme_root_uri()`
- `get_theme_root()`
- `get_theme_roots()`
- `get_stylesheet_directory()`
- `get_template_directory()`

Site Home

- `home_url()`
- `get_home_path()`

WordPress

- `admin_url()`
- `site_url()`
- `content_url()`
- `includes_url()`
- `wp_upload_dir()`

Multisite

- `get_admin_url()`
- `get_home_url()`

- `get_site_url()`
- `network_admin_url()`
- `network_site_url()`
- `network_home_url()`

Constants

WordPress makes use of the following constants when determining the path to the content and plugin directories. These should not be used directly by plugins or themes, but are listed here for completeness.

```
WP_CONTENT_DIR  // no trailing slash, full paths only
WP_CONTENT_URL  // full url
WP_PLUGIN_DIR  // full path, no trailing slash
WP_PLUGIN_URL  // full url, no trailing slash

// Available per default in MS, not set in single site install
// Can be used in single site installs (as usual: at your own risk)
UPLOADS // (If set, uploads folder, relative to ABSPATH) (for e.g.: /wp-content/uploads)
```

Related

WordPress Directories

Function		Directory
`home_url()`	Home URL	`https://www.example.com`
`site_url()`	Site directory URL	`https://www.example.com` or `https://www.example.com/wordpress`
`admin_url()`	Admin directory URL	`https://www.example.com/wp-admin`
`includes_url()`	Includes directory URL	`https://www.example.com/wp-includes`
`content_url()`	Content directory URL	`https://www.example.com/wp-content`
`plugins_url()`	Plugins directory URL	`https://www.example.com/wp-content/plugins`
`wp_upload_dir()`	Upload directory URL (returns an array)	`https://www.example.com/wp-content/uploads`

Header Requirements

As described in Getting Started, the main PHP file should include a header comment that tells WordPress that the 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, 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 list 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/old-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: (*Important: never use for a plugin hosted in the WordPress.org Plugin Directory*)** Allows non-WordPress.org 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 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
 * Text Domain:       my-basics-plugin
 * Domain Path:       /languages
 */
```

Here's another example that 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:       https://www.gnu.org/licenses/gpl-2.0.txt
 */
```

Notes

[alert]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.[/alert]

Including a Software License

Most WordPress plugins are released under the GPL, which is the same license that WordPress itself uses. However, there are other compatible options available. It is always best to clearly indicate the license your plugin uses.

In the Header Requirements section, we briefly mentioned how you can indicate your plugin's license within the plugin header comment. Another common, and encouraged, practice is to place a license block comment near the top of your main plugin file (the same one that has the plugin header comment).

This license block comment usually looks something like this:

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

{Plugin Name} 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 {Plugin Name}. If not, see {URI to Plugin License}.
*/
```

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`

[alert]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().[/alert]

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" );
```

[info]In Multisite, looping through all blogs to delete options can be very resource intensive.[/info]