Notes On Building A WordPress Plugin

Anyone can build a plugin but it takes a lot of work to get the plugin done right and ready for sharing or WordPress review and approval.

I realized the extra work involved when I was looking at the result of creating a plugin scaffold using the WordPress Plugin Boilerplate Generator.

The idea of the generator, just like its downloadble sibling, is to provide a good starting point for creating a plugin.

There are other generators available, some of which may be better suited to smaller, single-purpose tasks like what I have in mind:

- 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

Yet generators and boilerplates are not the only ways to create a plugin. Code Generators, by their very nature, are overwhelming. They give you every single tool you might need and then they let you de-clutter your code, assuming you know how the code is supposed to work and whether your plugin will need all the pieces they give you. They also lock you into the way the boilerplate authors think things should be done, even if one size doesn't fit all.

This post will discuss some points that I've learned about building a plugin from a boilerplate and from scratch. This is not meant to be a tutorial on how to build a plugin but to cover some things developers need to keep in mind when working with WordPress plugins.

Organizing the code

The first thing you need to decide about your plugin project is how you will organize the code.

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

The architecture for the plugin

The architecture, or code organization, you choose for your plugin will likely depend on the size and complexity of your plugin since there is no one-size-fits-all architecture.

For small, single-purpose plugins that have limited interaction with WordPress core, themes, or other plugins, there's little benefit in engineering complex multi-file solutions. In that case, you can use one of the following models:

- Single plugin file, containing functions
- Single plugin file, containing a class, instantiated object and optionally functions

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.

- Main plugin file, then one or more class files

Activation / Deactivation Hooks

WordPress provides hooks for the activation and deactivation of plugins. As with everything else we discuss in this post, whether you use it or not depends on what your plugin does.

One example of when we use these hooks is when working with custom post types.

When the plugin containing Custom Post Types (CPTs) is first activated we want to make sure we run the function that creates the CPTs. We also want to flush the rewrite rules so that the new CPTs are available.

```php
<?php
/**
 * Activate the plugin.
 */
function rivendellweb_cpt_activate() {
  // function defined elsewhere
  rivendellweb_cpt_setup_book();
  flush_rewrite_rules();
}

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

Likewise, when we deactivate the plugin we want to remove the CPTs to make sure that we don't have any orphaned CPTS. We also want to flush the rewrite rules to make the CPTs are no longer available.

```php
<?php
function rivendellweb_cpt_deactivate() {
  unregister_post_type( 'book' );
  flush_rewrite_rules();
}
register_deactivation_hook( __FILE__, 'rivendellweb_cpt_deactivate' );
```

Activate and deactivate hooks are a good way to set up and tear down the elements that are created by the plugin.

Uninstalling the plugin

Uninstalling the plugin is not the same as deactivating it. Deactivating the plugin keeps the code available and you can activate it again if needed.

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

This table shows the actions that are available for deactivation and uninstall code.

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

There are two ways to run uninstall code for a plugin: Running an uninstall hook or creating an `uninstall`.php file at the root of the plugin directory.

Using the hook is simple. Create the function that will do what you need to do when uninstalling the plugin. Then call the register_uninstall_hook() function to register the function with WordPress.

```
<?php
function rivendellweb_uninstall() {
  $option_name = 'wpor'wporg_option'delete_option($option_name);

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

```
register_uninstall_hook(__FILE__, 'rivendellweb_uninstall');
"DROP TABLE IF EXISTS {$wpdb->prefix}mytable"'rivendellweb_uninstall'
```

The second way is to create an `uninstall`.php file at the root of the plugin directory. This file will take precedence over the uninstall hook.

In the `uninstall`.php file I'm less concerned about creating classes and functions. I'm ok with running procedural code.

Warning:

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()`.

```
<?php
if (!defined('WP_U'WP_UNINSTALL_PLUGIN'    die;
}

$option_name = 'wporg_option';

delete_option($option_name);'wporg_option'// drop a custom database tabl
$wpdb->query("DROP TABLE IF EXISTS {$wpdb->prefix}mytable");
"DROP TABLE IF EXISTS {$wpdb->prefix}mytable"
```

Prefix Everything And Code Defensively

Because the code for your plugin will run in the WordPress global space there is a good chance that names will conflict with WordPress Core or other plugins if you’re not careful.

I take two approaches to avoid conflicts: I prefix all functions, classes, and variables I use in my plugins with my domain name (rivendellweb), the name of the plugin, or a combination of the two.

If I’m using an item from another plugin or a function I will always check if they exist. How to check if an item exists depends on what we’re looking for:

- **Variables**
  - isset() (includes arrays, objects, etc.)
- **Functions**
  - function_exists()
- **Classes**
  - get_declared_classes() lists all classes known to the current PHP script (both builtin and loaded)
  - class_exists() checks if a class exists in the declared classes array
- **Constants**
  - defined()

Plugin Security

Because we can’t fully trust the content that we get from our users, we should always work in a secure way. WordPress provides a few tools to ensure the security of your plugin.

Check For User Capabilities

The most important step in creating an efficient security layer is having a user permission system in place. WordPress provides this in the form of User Roles and Capabilities.

- **User roles** is just a fancy way of saying which group the user belongs to. Each group has a specific set of predefined capabilities
- **User capabilities** are the specific permissions that you assign to each user role
  - Every user logged into WordPress is automatically assigned a set of capabilities based on their role

As you build a plugin, you can check what role the user has and whether their role is allowed to perform the task you want them to perform.

You can also check if the user has the capability to perform a given task and fail the task if they are not allowed to do it.

To check for a specific role you need a custom function like this:

```php
<?php
function rivendellweb_user_has_role($role_to_check){
    $user = wp_get_current_user();
    if(in_array( $role_to_check, (array) $user->roles )){
        return true;
    }
    return false;
}
```

and then call it in your code like this:

```php
<?php
if (
  rivendellweb_user_has_role( 'administrator' )
) {
  echo 'User is an administrator';
} else {
  echo 'User is not an administrator';
}
```

You can check if a user has a specific capability by using the builtin function current_user_can()

```php
<?php
if ( current_user_can( 'edit_posts' ) ) {
  echo 'User can edit posts';
} else {
  echo 'User cannot edit posts';
}
```

See Roles and Capabilities for a full list of all the roles and capabilities and Capability vs. Role Table for a list of what capabilities are assigned to each role by default. Plugins like Members allow you to expand the default capabilities of a user or group of users by adding or removing capabilities. I discussed the plugin in Expanding Teams and Roles in WordPress Using The Members Plugin.

Sanitizing Input

Not that we don't trust users but it's always better to trust but verify that there wasn't an honest mistake (or deliberate action) on the part of the user, editor, or administrator.

WordPress provides functions to sanitize user input. The functions are:

- sanitize_email()
- sanitize_file_name()
- sanitize_hex_color()
- sanitize_hex_color_no_hash()
- sanitize_html_class()
- sanitize_key()
- sanitize_meta()
- sanitize_mime_type()
- sanitize_option()
- sanitize_sql_orderby()
- sanitize_text_field()
- sanitize_textarea_field()
- sanitize_title()
- sanitize_title_for_query()
- sanitize_title_with_dashes()
- sanitize_user()
- esc_url_raw()
- wp_kses()
- wp_kses_post()

The idea is that whenever we get user input, whether a regular user or an administrator, we should make sure it won't break the system.

Let's assume that we have the following HTML code:

```html
<input id="title" type="text" name="title">
```

Before updating the database with the content of the post we should sanitize the input to make sure it’s safe.

```php
<?php
$title = sanitize_text_field( $_POST['title'] );
update_post_meta( $post->ID, 'title', $title );
```

Nonces

“Nonce” is a portmanteau of “Number used ONCE”. It is a unique hexadecimal serial number used to verify the origin and intent of requests for security purposes. As the name suggests, each nonce can only be used once.

Nonces are usual paired with capability checking. The capability check makes sure the user can perform the task and the nonce makes sure the user is who they say they are. Doing both in tandem means that data is only changing when the user expects it to be changing.

I’ve taken the following example from nonces. It only shows how to use wp_create_nonce() for the link to delete a post. Other functions in the example will verify the nonce using wp_verify_nonce() before performing the task.

```php
<?php
function wporg_generate_delete_link( $content ) {
  if ( is_single() && in_the_loop() && is_main_query() ) {
      // Add query arguments: action, post, nonce
      $url = add_query_arg(
        [
          'action' => 'wporg_frontend_delete',
          'post'    => get_the_ID(),
          'nonce'   => wp_create_nonce( 'wporg_frontend_delete' ),
        ], home_url()
   'action'     return $content . ' <a href="' . esc_url( $url ) . '">' .
  }

  return null;
}
```

Then, in the handler function for the delete link, we verify that we have the nonce and that it’s valid for the link we clicked.

```
<?php
function wporg_delete_post() {
  if ( isset( $_GET['acti'action'   && isset( $_GET['nonce'] )
    && 'wporg_frontend_delete' === $_GET['action']
    && w'nonce'y_nonce( $_GET['nonce'], 'wporg_frontend_delete' ) ) {

      'nonce'// rest of the function that actually deletes the post

  }
}
```