# Custom Hooks


- Create a Hook
- Add a Callback to the Hook
- Naming Conflicts
- Examples
  - Extensible Action: Settings Form
  - Extensible Filter: Custom Post Type


An important, but often overlooked practice is using custom hooks in your plugin so that other developers can extend and modify it.

Custom hooks are created and called in the same way that WordPress Core hooks are.

# Create a Hook

To create a custom hook, use `do_action()` for Actions and `apply_filters()` for Filters.

[info]We recommend using `apply_filters()` on any text that is output to the browser. Particularly on the frontend.

This makes it easier for plugins to be modified according to the user's needs.[/info]

# Add a Callback to the Hook

To add a callback function to a custom hook, use `add_action()` for Actions and `add_filter()` for Filters.

# Naming Conflicts

Naming conflicts ("collisions") occur when two developers use the same hook name for completely different purposes. This leads to difficult to find bugs. So it's important to prefix your hook names with a unique string to avoid hook name collisions.collisions with other plugins.

For example, a filter named `email_body` is generic enough that two or more developers could use this hook in different plugins for different purposes. So to avoid this, a prefix is added. For example, functions used as examples in this handbook use `wporg_` as the prefix.

When you choose your prefix, you can use your company name, your wp handle, the plugin name, anything you like really. The goal is to make it unique so choose wisely.

# Examples

## Extensible Action: Settings Form

If your plugin adds a settings form to the Administrative Panels, you can use Actions to allow other plugins to add their own settings to it.

```
do_action( 'wporg_after_settings_page_html' );
```

Now another plugin can register a callback function for the `wporg_after_settings_page_html` hook and inject new settings:

```
add_action( 'wporg_after_settings_page_html', 'myprefix_add_settings' );
```

Note that because this is an action, no value is returned. Also note that since no priority is given, it will run at default priority 10.

## Extensible Filter: Custom Post Type

In this example, when the new post type is registered, the parameters that define it are passed through a filter, so another plugin can change them before the post type is created.

```
function wporg_create_post_type() {
  $post_type_params = [/* ... */];

  register_post_type(
    'post_type_slug',
    apply_filters( 'wporg_post_type_params', $post_type_params )
```

```
  );
}
```

Now another plugin can register a callback function for the `wporg_post_type_params` hook and change post type parameters:

```
function myprefix_change_post_type_params( $post_type_params ) {
  $post_type_params['hierarchical'] = true;
  return $post_type_params;
}
add_filter( 'wporg_post_type_params', 'myprefix_change_post_type_params' );
```

Note that filters take data, modify it, and return it. So the code called ( `myprefix_change_post_type_params` ) doesn't output anything using echo or html, or anything else directly to the screen. Also note that the returned value is used directly by `register_post_type` without being assigned to a variable first. This is simple to skip that extra (an unnecessary) step.

Also note that since no priority is given, it will run at default priority 10. And since there is no value for the number of parameters expected, the default of one is assumed.

---

□□□ #2
□ □□□ □□□ 2026-10-26 15:37:07 UTC
□ □□□ □□□ 2026-10-26 15:37:07 UTC