# Advanced development functions

## 1) add_action();

add_action( *string* $tag, *callable* $function_to_add, *int* $ priority = 10, *int* $accepted_args = 1 )

Hooks a function on to a specific action.

Actions are the hooks that the WordPress core launches at specific points during execution, or when specific events occur. Plugins can specify that one or more of its PHP functions are executed at these points, using the Action API.

**Parameters :-**

**$tag**

*(string) (Required)* The name of the action to which the $function_to_add is hooked.

**$function_to_add**

*(callable) (Required)* The name of the function you wish to be called.

**$priority**

*(int) (Optional)* Used to specify the order in which the functions associated with a particular action are executed. Lower numbers correspond with earlier execution, and functions with the same priority are executed in the order in which they were added to the action.
*Default value: 10*

**$accepted_args**

*(int) (Optional)* The number of arguments the function accepts.
*Default value: 1*

**Return**

*(true)* Will always return true.

**Accepted Arguments**

A hooked function can optionally accept arguments from the action call, if any are set to be passed. In this simplistic example, the `echo_comment_id` function takes the `$comment_id` argument, which is automatically passed to when the `do_action()` call using the `comment_id_not_found` filter hook is run.

```
/**
 * Warn about comment not found
 *
 * @param int $comment_id Comment ID.
 */
function echo_comment_id( $comment_id ) {
    printf( 'Comment ID %s could not be found', esc_html( $comment_id ) );
}
add_action( 'comment_id_not_found', 'echo_comment_id', 10, 1 );
```

## 2) add_filter()

add_filter( *string* $tag, *callable* $function_to_add, *int* $ priority = 10, *int* $accepted_args = 1 )

Hook a function or method to a specific filter action.

### Description:-

WordPress offers filter hooks to allow plugins to modify various types of internal data at runtime.

A plugin can modify data by binding a callback to a filter hook. When the filter is later applied, each bound callback is run in order of priority, and given the opportunity to modify a value by returning a new value.

The following example shows how a callback function is bound to a filter hook.Note that `$example` is passed to the callback, (maybe) modified, then returned:

```
function example_callback( $example ) {
    // Maybe modify $example in some way.
    return $example;
}
add_filter( 'example_filter', 'example_callback' );
```

**Parameters**

**$tag**

*(string) (Required)* The name of the filter to hook the $function_to_add callback to.

**$function_to_add**

*(callable) (Required)* The callback to be run when the filter is applied.
**$priority**

*(int) (Optional)* Used to specify the order in which the functions associated with a particular action are executed. Lower numbers correspond with earlier execution, and functions with the same priority are executed in the order in which they were added to the action.
*Default value: 10*
**$accepted_args**

*(int) (Optional)* The number of arguments the function accepts.
*Default value: 1*

Return
*(true)*

- Hooked functions can take extra arguments that are set when the matching do_action() or apply_filters() call is run. For example, the comment_id_not_found action will pass the comment ID to each callback.
- Although you can pass the number of `$accepted_args`, you can only manipulate the `$value`. The other arguments are only to provide context, and their values cannot be changed by the filter function.
- You can also pass a class method as a callback.

**Example:** Let's add extra sections to TwentySeventeen Front page.
*By default, TwentySeventeen theme has 4 sections for the front page.* ***This example will make them 6***

```
add_filter( 'twentyseventeen_front_page_sections',
               'prefix_custom_front_page_sections' );

function prefix_custom_front_page_sections( $num_sections )
{
         return 6;
}
```

# 3)add_shortcord()

## Description

Adds a hook for a shortcode tag.

## Usage

```
<?php add_shortcode( $tag , $func ); ?>
```

## Parameters

**`$tag`**

(*string*) (*required*) Shortcode tag to be searched in post content
Default: *None*

**`$func`**

(*callable*) (*required*) Hook to run when shortcode is found
Default: *None*

The **Shortcode API** is a simple set of functions for creating WordPress shortcodes for use in posts and pages. For instance, the following shortcode (in the body of a post or page) would add a photo gallery of images attached to that post or page: `[gallery]`
The API enables plugin developers to create special kinds of content (e.g. forms, content generators) that users can attach to certain pages by adding the corresponding shortcode into the page text.

The Shortcode API makes it easy to create shortcodes that support attributes like this:

```
[gallery id="123" size="medium"]
```

The API handles all the tricky parsing, eliminating the need for writing a custom regular expression for each shortcode. Helper functions are included for setting and fetching default attributes. The API supports both self-closing and enclosing shortcodes.

As a quick start for those in a hurry, here's a minimal example of the PHP code required to create a shortcode:

```
//[foobar]

function foobar_func( $atts ){
        return "foo and bar";

}

add_shortcode( 'foobar', 'foobar_func' );
```

This will create [foobar] shortcode that returns as: foo and bar

With attributes:

```
// [bartag foo="foo-value"]
function bartag_func( $atts ) {
    $a = shortcode_atts( array(
        'foo' => 'something',
        'bar' => 'something else',
    ), $atts );

    return "foo = {$a['foo']}";
}
add_shortcode( 'bartag', 'bartag_func' );
```

This creates a "[bartag]" shortcode that supports two attributes: ["foo" and "bar"]. Both attributes are optional and will take on default options [foo="something" bar="something else"] if they are not provided. The shortcode will return as foo = {the value of the foo attribute}.

## 4) do_shortcode()

do_shortcode( *string* $content, *bool* $ignore_html = false )

Search content for shortcodes and filter shortcodes through their hooks.

### Description

If there are no shortcode tags defined, then the content will be returned without any filtering. This might cause issues when plugins are disabled but the shortcode will still show up in the post or content.

Parameters

**$content**

*(string) (Required)* Content to search for shortcodes.

**$ignore_html**

*(bool) (Optional)* When true, shortcodes inside HTML elements will be skipped.
*Default value: false*

**Return**

*(string)* Content with shortcodes filtered out.

```
// Use shortcode in a PHP file (outside the post editor).
echo do_shortcode( '' );
```

```
// In case there is opening and closing shortcode.
echo do_shortcode( '[iscorrect]' . $text_to_be_wrapped_in_shortcode . '[/iscorrect]' );
```

```
// Enable the use of shortcodes in text widgets.
add_filter( 'widget_text', 'do_shortcode' );
```

```
// Use shortcodes in form like Landing Page Template.
echo do_shortcode( '[contact-form-7 id="91" title="quote"]' );
```

```
// Store the short code in a variable.
$var = do_shortcode( '' );
echo $var;
```

The most common use of the shortcode is the following

```
<?php echo do_shortcode('[name_of_shortcode]'); ?>
```

If there are no shortcode tags defined, then the content will be returned without any filtering. This might cause issues if a plugin is disabled as its shortcode will still show up in the post or content.

## 5)register_nav_menu()

### Description

Registers a *single* custom Navigation Menu in the custom menu editor (in WordPress 3.0 and above). This allows administration users to create custom menus for use in a theme.
See register_nav_menus() for creating multiple menus at once.

### Usage

```
<?php register_nav_menu( $location, $description ); ?>
```

### Parameters

**`$location`**

(*string*) (*required*) Menu location identifier, like a slug.
Default: *None*

**`$description`**

(*string*) (*required*) Menu description - for identifying the menu in the dashboard.
Default: *None*

### Return Values

None.

### Examples

```
<?php
add_action( 'after_setup_theme', 'register_my_menu' );
function register_my_menu() {
  register_nav_menu( 'primary', __( 'Primary Menu', 'theme-slug' ) );
}
?>
```

### Notes

- This function automatically registers custom menu support for the theme therefore you do **not** need to call `add_theme_support( 'menus' );`
- This function actually works by simply calling register_nav_menus() in the following way:

```
register_nav_menus( array( $location => $description ) );
```

## *What is Custom Post Type in WordPress?*

Custom post types are content types like posts and pages. Since WordPress evolved from a simple blogging platform into a robust CMS, the term post stuck to it. However, a post type can be any kind of content. By default,

### 1) `register_post_type()`

First we will show you a quick and fully working example so that you understand how it works. Take a look at this code:

```
// Our custom post type function
function create_posttype() {

    register_post_type( 'movies',
    // CPT Options
        array(
            'labels' => array(
                'name' => __( 'Movies' ),
                'singular_name' => __( 'Movie' )
            ),
            'public' => true,
            'has_archive' => true,
            'rewrite' => array('slug' => 'movies'),
        )
    );
}
// Hooking up our function to theme setup
add_action( 'init', 'create_posttype' );
```

What this code does is that it registers a post type `'movies'` with an array of arguments. These arguments are the options of our custom post type. This array has two parts, the first part is labels, which itself is an array. The second part contains other arguments like public visibility, has archive, and slug that will be used in URLs for this post type.

## Usage

```
<?php register_post_type( $post_type, $args ); ?>
```

## Parameters

**`$post_type`**

(*string*) (*required*) Post type. (**max. 20 characters, cannot contain capital letters or spaces**)

Default: *None*

**`$args`**

(*array*) (*optional*) An array of arguments.

Default: *None*

## Arguments

**`label`**

(*string*) (*optional*) A **plural** descriptive name for the post type marked for translation.

Default: Value of $labels['name']

**`labels`**

(*array*) (*optional*) labels - An array of labels for this post type. By default, post labels are used for non-hierarchical post types and page labels for hierarchical ones.

Default: if empty, 'name' is set to value of 'label', and 'singular_name' is set to value of 'name'.

- **'name'** - general name for the post type, usually plural. The same and overridden by $post_type_object->label. Default is Posts/Pages
- **'singular_name'** - name for one object of this post type. Default is Post/Page
- **'add_new'** - the add new text. The default is "Add New" for both hierarchical and non-hierarchical post types. When internationalizing this string, please use a gettext context matching your post type. Example: `_x('Add New', 'product');`
- **'add_new_item'** - Default is Add New Post/Add New Page.
- **'edit_item'** - Default is Edit Post/Edit Page.
- **'new_item'** - Default is New Post/New Page.
- **'view_item'** - Default is View Post/View Page.
- **'view_items'** - Label for viewing post type archives. Default is 'View Posts' / 'View Pages'.
- **'search_items'** - Default is Search Posts/Search Pages.
- **'not_found'** - Default is No posts found/No pages found.
- **'not_found_in_trash'** - Default is No posts found in Trash/No pages found in Trash.
- **'parent_item_colon'** - This string isn't used on non-hierarchical types. In hierarchical ones the default is 'Parent Page:'.
- **'all_items'** - String for the submenu. Default is All Posts/All Pages.

- **'archives'** - String for use with archives in nav menus. Default is Post Archives/Page Archives.
- **'attributes'** - Label for the attributes meta box. Default is 'Post Attributes' / 'Page Attributes'.
- **'insert_into_item'** - String for the media frame button. Default is Insert into post/Insert into page.
- **'uploaded_to_this_item'** - String for the media frame filter. Default is Uploaded to this post/Uploaded to this page.
- **'featured_image'** - Default is Featured Image.
- **'set_featured_image'** - Default is Set featured image.
- **'remove_featured_image'** - Default is Remove featured image.
- **'use_featured_image'** - Default is Use as featured image.
- **'menu_name'** - Default is the same as `name`.
- **'filter_items_list'** - String for the table views hidden heading.
- **'items_list_navigation'** - String for the table pagination hidden heading.
- **'items_list'** - String for the table hidden heading.
- **'name_admin_bar'** - String for use in New in Admin menu bar. Default is the same as `singular_name`.

## 2) register_taxonomy()

### Description

This function adds or overwrites a taxonomy. It takes in a name, an object name that it affects, and an array of parameters. It does not return anything.

Care should be used in selecting a taxonomy name so that it does not conflict with other taxonomies, post types, and reserved WordPress public and private query variables. A complete list of those is described in the Reserved Terms section. In particular, capital letters should be avoided (This was allowed in 3.0, but not enforced until 3.1 with the "Cheatin'" error).

### Usage

```
<?php register_taxonomy( $taxonomy, $object_type, $args ); ?>
```

**Use the `init` action to call this function.** Calling it outside of an action can lead to troubles. See #15568 for details.

Better **be safe than sorry** when registering custom taxonomies for custom post types. Use register_taxonomy_for_object_type() right after the function to interconnect them. Else you could run into minetraps where the post type isn't attached inside filter callback that run during `parse_request` or `pre_get_posts`.

## Parameters

**`$taxonomy`**

(*string*) (*required*) The name of the taxonomy. Name should only contain lowercase letters and the underscore character, and not be more than 32 characters long (database structure restriction).

Default: *None*

**`$object_type`**

(*array/string*) (*required*) Name of the object type for the taxonomy object. Object-types can be built-in Post Type or any Custom Post Type that may be registered.

Default: *None*

**`label`**

(*string*) (*optional*) A **plural** descriptive name for the taxonomy marked for translation.

Default: overridden by *$labels->name*

**`labels`**

(*array*) (*optional*) labels - An array of labels for this taxonomy. By default tag labels are used for non-hierarchical types and category labels for hierarchical ones.

Default: if empty, name is set to label value, and singular_name is set to name value

- **'name'** - general name for the taxonomy, usually plural. The same as and overridden by $tax->label. Default is `_x( 'Post Tags', 'taxonomy general name' )` or `_x( 'Categories', 'taxonomy general name' )`. When internationalizing this string, please use a gettext context matching your post type. Example: `_x('Writers', 'taxonomy general name');`
- **'singular_name'** - name for one object of this taxonomy. Default is `_x( 'Post Tag', 'taxonomy singular name' )` or `_x( 'Category', 'taxonomy singular name' )`. When internationalizing this string, please use a gettext context matching your post type. Example: `_x('Writer', 'taxonomy singular name');`
- **'menu_name'** - the menu name text. This string is the name to give menu items. If not set, defaults to value of *name* label.
- **'all_items'** - the all items text. Default is `__( 'All Tags' )` or `__( 'All Categories' )`
- **'edit_item'** - the edit item text. Default is `__( 'Edit Tag' )` or `__( 'Edit Category' )`
- **'view_item'** - the view item text, Default is `__( 'View Tag' )` or `__( 'View Category' )`
- **'update_item'** - the update item text. Default is `__( 'Update Tag' )` or `__( 'Update Category' )`
- **'add_new_item'** - the add new item text. Default is `__( 'Add New Tag' )` or `__( 'Add New Category' )`
- **'new_item_name'** - the new item name text. Default is `__( 'New Tag Name' )` or `__( 'New Category Name' )`
- **'parent_item'** - the parent item text. This string is not used on non-hierarchical taxonomies such as post tags. Default is null or `__( 'Parent Category' )`

- **'parent_item_colon'** - The same as `parent_item`, but with colon `:` in the end null, `__( 'Parent Category:' )`
- **'search_items'** - the search items text. Default is `__( 'Search Tags' )` or `__( 'Search Categories' )`
- **'popular_items'** - the popular items text. This string is not used on hierarchical taxonomies. Default is `__( 'Popular Tags' )` or null
- **'separate_items_with_commas'** - the separate item with commas text used in the taxonomy meta box. This string is not used on hierarchical taxonomies. Default is `__( 'Separate tags with commas' )`, or null
- **'add_or_remove_items'** - the add or remove items text and used in the meta box when JavaScript is disabled. This string is not used on hierarchical taxonomies. Default is `__( 'Add or remove tags' )` or null
- **'choose_from_most_used'** - the choose from most used text used in the taxonomy meta box. This string is not used on hierarchical taxonomies. Default is `__( 'Choose from the most used tags' )` or null
- **'not_found'** (3.6+) - the text displayed via clicking 'Choose from the most used tags' in the taxonomy meta box when no tags are available and (4.2+) - the text used in the terms list table when there are no items for a taxonomy. Default is `__( 'No tags found.' )` or `__( 'No categories found.' )`

## Example Private Taxonomy

If you do not want your taxonomy to be exposed publicly, you can use the 'public' and 'rewrite' parameters to suppress it. It will be available to use internally by your plugin or theme, but will not generate a url of it's own.

```
<?php
add_action( 'init', 'create_private_book_tax' );
function create_private_book_tax() {
    register_taxonomy(
        'genre',
        'book',
        array(
            'label' => __( 'Genre' ),
            'public' => false,
            'rewrite' => false,
            'hierarchical' => true,
        )
    );}

?>
```

# Widget Area

## 1) register_sidebar()

### Description

Builds the definition for a single sidebar and returns the ID. Call on "widgets_init" action.

Accepts either a string or an array and then parses that against a set of default arguments for the new sidebar. WordPress will automatically generate a sidebar ID and name based on the current number of registered sidebars if those arguments are not included.

When allowing for automatic generation of the name and ID parameters, keep in mind that the incrementor for your sidebar can change over time depending on what other plugins and themes are installed.

If theme support for 'widgets' has not yet been added when this function is called, it will be automatically enabled through the use of add_theme_support()

### Usage

```
<?php register_sidebar( $args ); ?>
```

### Default Usage

```
<?php $args = array(
        'name'          => __( 'Sidebar name', 'theme_text_domain' ),
        'id'            => 'unique-sidebar-id',
        'description'   => '',
        'class'         => '',
        'before_widget' => '<li id="%1$s" class="widget %2$s">',
        'after_widget'  => '</li>',
        'before_title'  => '<h2 class="widgettitle">',
        'after_title'   => '</h2>' ); ?>
```

### Parameters

**args**

(*string/array*) (*optional*) Builds Sidebar based off of 'name' and 'id' values.

Default: *None*

- `name` - Sidebar name (default is localized 'Sidebar' and numeric ID).
- `id` - Sidebar id - Must be all in lowercase, with no spaces (default is a numeric auto-incremented ID). If you do not set the `id`argument value, you will get `E_USER_NOTICE` messages in debug mode, starting with version 4.2.
- `description` - Text description of what/where the sidebar is. Shown on widget management screen. (Since 2.9) (default: empty)
- `class` - CSS class to assign to the Sidebar in the Appearance -> Widget admin page. This class will only appear in the source of the WordPress Widget admin page. It will not be included in the frontend of your website. Note: The value `"sidebar"` will be prepended to the class value. For example, a class of `"tal"` will result in a class value of `"sidebar-tal"`. (default: empty).
- `before_widget` - HTML to place before every widget(default: `<li id="%1$s" class="widget %2$s">`) Note: uses sprintf for variable substitution
- `after_widget` - HTML to place after every widget (default: `</li>\n`).
- `before_title` - HTML to place before every title (default: `<h2 class="widgettitle">`).
- `after_title` - HTML to place after every title (default: `</h2>\n`).

- This example creates a sidebar named “Main Sidebar” with and before and after the title.

```

/**
 * Add a sidebar.
 */
function wpdocs_theme_slug_widgets_init() {
    register_sidebar( array(
        'name'      => __( 'Main Sidebar', 'textdomain' ),
        'id'       => 'sidebar-1',
'description'  => __( 'Widgets in this area will be shown on all posts and pages.',
        'before_widget' => '<li id="%1$s" class="widget %2$s">',
        'after_widget'  => '</li>',
        'before_title'  => '<h2 class="widgettitle">',
        'after_title'  => '</h2>',
    ) );
}
add_action( 'widgets_init', 'wpdocs_theme_slug_widgets_init' );
```

## 2) **dynamic_sidebar**

dynamic_sidebar( *int|string* $index = 1 )

Display dynamic sidebar.

## Description

By default this displays the default sidebar or ‘sidebar-1’. If your theme specifies the ‘id’ or ‘name’ parameter for its registered sidebars you can pass an id or name as the $index parameter. Otherwise, you can pass in a numerical index to display the sidebar at that index.

**Parameters**

**$index**

*(int|string) (Optional)* Index, name or ID of dynamic sidebar.
*Default value: 1*

**Return**

*(bool)* True, if widget sidebar was found and called. False if not found or not called

Here is the recommended use of this function:

```
<?php if ( is_active_sidebar( 'left-sidebar' ) ) : ?>
    <ul id="sidebar">
        <?php dynamic_sidebar( 'left-sidebar' ); ?>
    </ul>
<?php endif; ?>
```

```
<ul id="sidebar">
    <?php dynamic_sidebar( 'right-sidebar' ); ?>
</ul>
```

```
<ul id="sidebar">
<?php if ( ! dynamic_sidebar() ) : ?>
    <li>{static sidebar item 1}</li>
    <li>{static sidebar item 2}</li>
<?php endif; ?>
</ul>
```

Example2

```
<div class=”sidebar new-sidebar”>
        <?php if( is_active_sidebar(‘new_sidebar’)); ?>
                <?php dynamic_sidebar(‘new_sidebar’); ?>
<?php else : ?>
        <!---------code-------->
<?php elseif; ?>
</div>
```