

Day 1

Plugin and Widget Development

**BY:**

**BRUNO ANGELO MEDEIROS**bruno.medeiros@sc.senai.br

# CREATE PLUGIN

## FILES AND FOLDERS

To our first step, we’ll need to create our basic files and folder structure. WordPress store its plugins in the **wp-content/plugin/** folder. This is the place where we’ll be adding our files as well.

Now create a new file inside and name it **first\_plugin.php**. This will be our main plugin file and its name should be the same as our plugin’s slug.

Open our plugin’s main file and paste in the following code:

1. <?php
2. /\*\*
3. \* Plugin Name: My First Plugin
4. \* Plugin URI: http://yourdomain.com
5. \* Description: Creating my first plugin
6. \* Version: 1.0.0
7. \* Author: Bruno Medeiros
8. \* Author URI: http://yourdomain.com
9. \* License: GPL2
10. \*/

This comment won’t be visible directly in the WordPress admin. WordPress does use the data within it to output the plugin’s name and some other data in the Plugins section of the backend.

# HOW HOOKS WORKS

## HOOKS

To create plugin we need to use hooks, that allows we to do two things, change default functionality within WordPress and add our own functionality without modifying core WordPress files at all, using this two types of: **actions** and **filters**.

### THE ANATOMY OF A HOOK

When we use hooks we generally use two functions: **add\_action()** and **add\_filter()**. Both these function take four parameters:

1. The first parameter is the **tag**. This is tells WordPress where to hook our function, when it should be executed.
2. The second parameter should be the **name of the hooked function**.
3. The third parameter is **the priority of the hook**. This determines the order in which it is executed if multiple functions are hooked into the same tag.
4. The fourth parameter **defines the number of parameters passed to this function**. By default this is 1, but some tags can have more.

Using the plugin example above, we need create a hooked function. In the **first\_plugin.php** file, paste the following:

1. add\_action( 'wp\_insert\_post', 'email\_post\_author', 10, 3 );
2. **function** email\_post\_author( $post\_id, $post, $update ) {
3. $email = 'mymail@mail.com';
4. $subject = 'New Post Published';
5. $message = 'A new post was published, use this link to view it: ' . get\_permalink( $post->ID );
6. wp\_mail( $email, $subject, $message );
7. }

We’re using the **add\_action()** function to let WordPress know that we’d like to hook a function into **wp\_insert\_post**.

### ACTIONS AND FILTERS

Action it’s a piece of code that is executed at a specific time, and a filter can be used to modify content before WordPress uses it. We could use a filter to modify the “your password is incorrect” text on the login page. Here’s how:

1. add\_filter( 'login\_errors', 'modify\_login\_errors' );
2. **function** modify\_login\_errors() {
3. **return** 'Login unsuccessful, try again';
4. }

By implementing this change we’ve added a bit more security to our login process and we can also control the text.

### HOOKS, ACTIONS, FILTERS, HOOKED FUNCTION AND TAGS

Summing up, **Hook** is an umbrella term for actions and filters. **Actions** allow us to add our own functionality next to existing functionality. **Filters** allow us to modify existing functionality.

# ANOTHER EXAMPLE

For example, if our site is for an organization and each of the organization's functions has a top level page with subpages for individual departments, then we'll want to list all of those departments along with a link to the top level function page. We'll want to do this on each of those department pages and on any subpages they might have as well as with the top level page itself.

To do this, we won't be able to output lists of related content by running queries on the current post type or taxonomy term. Instead, we'll have to identify where in the structure the current page is and then display a list of links accordingly.

Here, we'll create this, by creating a function we can add to our sidebar file or perhaps above the content in our template files or activate via a hook.

This consists of two stages:

* identifying where the current page is in the structure
* outputting a list of pages

## CREATING THE PLUGIN

So our first step is to create a plugin file called **list-subpages.php**.

Open our plugin file and add the following:

1. <?php
2. /\*\*
3. \* Plugin Name: List Subpages
4. \* Description: This plugin checks if the current page has parent or child pages and if so, outputs a list of the highest ancestor page and its descendants. It creates a function called list\_subpages() which you insert into your theme or activate via a hook to work.
5. \* Version: 1.0
6. \* Author: Russian Training
7. \* Author URI: http://yourdomain.com
8. \* License: GPLv2
9. \*/

## IDENTIFYING THE CURRENT PAGE IN THE HIERARCHY

To find out where the current page is in the page hierarchy, we need to do four things:

1. Check that this is actually a page
2. Check if this post has parents
3. If not, then we know it's the top level ancestor for this part of the hierarchy
4. If so, we need to identify the top level ancestor using [**get\_post\_ancestors()**](http://codex.wordpress.org/Function_Reference/get_post_ancestors)

Create a new function with a conditional tag in it to check we're on a page:

1. **function** check\_for\_page\_tree() {
3. //start by checking if we're on a page
4. **if**( is\_page() ) {
6. }
8. }

Inside the **is\_page()** conditional tag, start by defining the **$post global** variable:

1. **function** check\_for\_page\_tree() {
3. //start by checking if we're on a page
4. **if**( is\_page() ) {
6. **global** $post;
8. }
10. }

Next we need to identify whether the current page has parents, which we do using **if( &post->post\_parent )**:

1. **function** check\_for\_page\_tree() {
3. //start by checking if we're on a page
4. **if**( is\_page() ) {
6. **global** $post;
8. // next check if the page has parents
9. **if** ( $post->post\_parent ) {
11. }
12. }
14. }

If the page does have ancestors, we need to identify the top-most of these, which we do using **get\_post\_ancestors():**

1. **function** check\_for\_page\_tree() {
3. //start by checking if we're on a page
4. **if**( is\_page() ) {
6. **global** $post;
8. // next check if the page has parents
9. **if** ( $post->post\_parent ){
11. // fetch the list of ancestors
12. $parents = array\_reverse( get\_post\_ancestors( $post->ID ) );
14. // get the top level ancestor
15. **return** $parents[0];
17. }
18. }
20. }

This defines a new variable of **$parents** whose value is the ID of the topmost page in the current branch of the hierarchy. The line **return $parents[0];** outputs that value so that we can use it in a later function.

Finally, we need to define what happens if the current page doesn't have parents, i.e. if it's the topmost ancestor itself. In that case we want to output the current page's ID, so we add the following to our function:

1. **return** $post->ID;

The entire funtion:

1. **function** check\_for\_page\_tree() {
3. //start by checking if we're on a page
4. **if**( is\_page() ) {
6. **global** $post;
8. // next check if the page has parents
9. **if** ( $post->post\_parent ) {
11. // fetch the list of ancestors
12. $parents = array\_reverse( get\_post\_ancestors( $post->ID ) );
14. // get the top level ancestor
15. **return** $parents[0];
17. }
18. }
20. **return** $post->ID;
22. }

## OUTPUTTING A LIST OF SUBPAGES

We use [**get\_pages()**](http://codex.wordpress.org/Function_Reference/get_pages) to identify the child pages of the page whose ID we’ve identified. We'll also need to output a link to the ancestor page at the beginning of the list.

### USING LIST\_PAGES() TO IDENTIFY SUBPAGES

Start by creating a new function with a check that we're on a page called **list\_subpages**.

The first thing we need to do inside is pull in the page ID we identified in the **check\_for\_page\_tree()** function, which we do with this line of code:

1. **function** list\_subpages() {
3. $ancestor = check\_for\_page\_tree();
4. }

Below that, define the arguments for the **get\_pages()** function:

Let's have a quick look at the arguments I've used:

1. $args = **array**(
2. 'child\_of' => $ancestor,
3. 'depth' => '-1',
4. 'title\_li' => ''
5. );

* **'child\_of' => $ancestor** identifies those pages which are a child of the **$ancestor** page
* **'depth' => '-1'** tells the function to go to as many levels in the hierarchy as there are in the site. We can change this if wejust want to display one or two levels.
* **'title\_li' => ''** ensures that what's output isn't wrapped in any HTML tags.

Next, we need to set the **list\_pages** variable:

1. $list\_pages = get\_pages( $args );

### OUTPUTTING THE LIST OF PAGES

Now that we have our pages, we need to output them with links. To do this, first check that **$list\_pages** hasn't returned an empty array and inside that check, the first link is to the top level page:

1. **if** ( $list\_pages ) {
2. ?>
3. <ul **class**="page-tree">
4. <li **class**="ancestor">
5. <a href="<?= get\_permalink( $ancestor ); ?>"><?= get\_the\_title( $ancestor ); ?></a>
6. </li>
7. </ul>
8. }

And then below that first <li> element but still inside the <ul>, use the **wp\_list\_pages()** function to output a list of the pages wrapped in hyperlinks to them. This will display a list of the page titles as links:

1. wp\_list\_pages( $args );

Our entire **list\_subpages()** function will now look like this:

1. **function** list\_subpages() {
3. // run the check\_for\_page\_tree function to fetch top level page
4. $ancestor = check\_for\_page\_tree();
6. // set the arguments for children of the ancestor page
7. $args = **array**(
8. 'child\_of' => $ancestor,
9. 'depth' => '-1',
10. 'title\_li' => ''
11. );
13. // set a value for get\_pages to check if it's empty
14. $list\_pages = get\_pages( $args );
16. // check if $list\_pages has value
17. **if** ( $list\_pages ) {
19. // open a list with the ancestor page at the top
20. ?>
22. <ul **class**="page-tree">
23. <?php // list ancestor page ?>
24. <li **class**="ancestor">
25. <a href="<?= get\_permalink( $ancestor ); ?>"><?= get\_the\_title( $ancestor ); ?></a>
26. </li>
28. <?php
29. // use wp\_list\_pages to list subpages of ancestor or current page
30. wp\_list\_pages( $args );
32. // close the page-tree list
33. ?>
34. </ul>
36. <?php
37. }
38. }

### ACTIVATING THE FUNCTION

We can activate the function in one of two ways:

* By calling **list\_subpages()** in one of our theme's template files, such as the sidebar.php file
* By attaching it to a hook in our theme.

# MAKING THE WIDGET

## CREATE

We create a new WordPress widgets by adding code to one of two places:

* A custom plugin, which we sholud use if we want to use the widgets with more than one theme or on more than one website.
* The **functions.php** file of the active theme – which should be either a child theme or a completely custom theme.

We extend the **WP\_Widget** class, which include some functions:

* a function to process the widgets
* a function to display a form for the widgets in the Widgets dashboard screen
* a function enabling widgets settings to be updated by users
* and a function to output the widgets in na widgets area it’s added to.

And use the **register\_widget()** function to register the widgets we’ve created.

### WIDGET API

The Widgets API includes the function we’ll need to create our widgets.

Firstly, there are four widget functions:

* **is\_active\_widget()**: a condicional tag which cheques whether an individual widgets is active. Don’t confuse it **with is\_active\_sidebar()**, which checks if widgets have benn added to a specific widgets area.
* **the\_widget()**: a template tag which displays a widgets outside of widgets areas.
* **register\_widget()**: the function to register a widgets.
* **unregister\_widget()**: unregister a widgets, meaning that it’s n longet available for users via the widgets screen.

There are also five internal functions:

* **wp\_register\_widget\_control()**: creates the controls on the Widgets screen so that users can amend the widget’s settings.
* **wp\_unregister\_widget\_control():** registers the widgets control that has been registered via **wp\_register\_widget\_control().**
* **wp\_convert\_widget\_settings()**: this converts a widget’s setting form single instance to multi-widget.
* **wp\_get\_widget\_defaults()**: core function, not to be used by plugin or theme developers
* **wp\_widget\_description()**: creates a widgets description that will be displayed on the Widgets screen.

To access these internal functions, we’ll make use of the **WP\_Widget** class. This is a constructor class, which we can extend to create additional widgets.

## CODE AND REGISTER

### SETTING UP

Create a new file in our **wp-content/plugins** directory. For this example we’ll call **list-subpages-widget.php.**

Add the following to the file:

1. <?php
2. /\*\*
3. \* Plugin Name: List Subpages Widget
4. \* Description: This widget checks if the current page has parent or child pages and if so, outputs a list of the highest ancestor page and its descendants.
5. \* Version: 0.1
6. \* Author: John Doe
7. \* Author URI: http://www.worldskills.org
8. \* License: GPLv2
9. \*/
10. ?>

With this, Wordpress can display the plugin in the Plugins screen.

### CREATING WIDGET CLASS

Now, we need create a new class to extend the **WP\_Widget** class.

Using this code into our plugin file:

1. **class** List\_Pages\_Widget **extends** WP\_Widget {
3. **function** \_\_construct() {
5. }
7. **function** form( $instance ) {
9. }
11. **function** update( $new\_instance, $old\_instance ) {
13. }
15. **function** widget( $args, $instance ) {
17. }
18. }

The class consists of:

* the **\_\_construct** function will construct the function. Inside that function we’ll define things like the ID of the widgets, its title and description.
* the **form** function will create the form in the Widgets screen that lets users customize or activate the widgets.
* the **update** function ensures that WordPress updates any setting that users input in the Widgets screen.
* the **widgets** function defines what’s output by the widgets on the front end of the site.

### REGISTERING

Our widgets won’t work unless we register it with WordPress. Below our class, add the function and hook to do this:

1. <?php
2. **function** register\_list\_pages\_widget() {
4. register\_widget( 'List\_Pages\_Widget' );
6. }
7. add\_action( 'widgets\_init', 'register\_list\_pages\_widget' );
8. ?>

The **register\_widget()** function is a WordPress function whose single parameter is the name of the class we just created.

We then hook our funcion to the **widgets\_init** hook to ensure that it is picker up by Wordpress.

## CONSTRUCT

To continues we need populate the **\_\_construct()** function which we created inside our **List\_Pages\_Widget** class.

So, edit the plugin file and edit it so it reads as follows:

1. **function** \_\_construct() {
3. parent::\_\_construct(
5. // base ID of the widget
6. 'list\_pages\_widget',
8. // name of the widget
9. \_\_('List Related Pages', 'worldskills' ),
11. // widget options
12. **array** (
13. 'description' => \_\_( 'Identifies where the current page is in the site structure and displays a list of pages in the same section of the site. Only works on Pages.', 'worldskills' )
14. )
16. );
18. }

This defines the parameters to create our widget. They are:

* the unique ID of the widget
* the name of the widget as seen on the Widgets screen
* an array of options including the description, which is displayed on the Widgets screen. This needs to explain to users what the widget will do.

### FORM

To create the form for our widgets, we’ll need to populate the **form** function that we’ve already added to our **List\_Pages\_Widget** class.

Add the follow code in the function:

1. **function** form( $instance ) {
3. $defaults = **array**(
4. 'depth' => '-1'
5. );
6. $depth = $instance[ 'depth' ];
8. // markup for form ?>
9. <p>
10. <label **for**="<?php echo $this->get\_field\_id( 'depth' ); ?>">Depth of list:</label>
11. <input **class**="widefat" type="text" id="<?php echo $this->get\_field\_id( 'depth' ); ?>" name="<?php echo $this->get\_field\_name( 'depth' ); ?>" value="<?php echo esc\_attr( $depth ); ?>">
12. </p>

15. }

We'll find that the widget now has a form now. But we need to allow the form to save what’s input.

### UPDATE

To allow to save, we need to work with the **update** function we created earlier on.

So, we need to edit with the code below:

1. **function** update( $new\_instance, $old\_instance ) {
3. $instance = $old\_instance;
4. $instance[ 'depth' ] = strip\_tags( $new\_instance[ 'depth' ] );
5. **return** $instance;
7. }

This replaces the old value of the depth field ($old\_instance) with the new value ($new\_instance), and sanitises it using **strip\_tags**. Now when we type into the field, it will save.

## OUTPUT

## EDITING THE FUNCTION WIDGET

We’ll need to edit the empty **widget** function we created earlier, in our plugin file. Start by defining the variable based on the form’s input:

1. **function** widget( $args, $instance ) {
3. extract( $args );
4. echo $before\_widget;
5. echo $before\_title . 'In this section:' . $after\_title;
7. }

Next, add our query and its output, editing the function so it reads like this:

1. **function** widget( $args, $instance ) {
3. // kick things off
4. extract( $args );
5. echo $before\_widget;
6. echo $before\_title . 'In this section:' . $after\_title;
8. // run a query if on a page
9. **if** ( is\_page() ) {
11. // run the check\_for\_page\_tree function to fetch top level page
12. $ancestor = check\_for\_page\_tree();
14. // set the arguments for children of the ancestor page
15. $args = **array**(
16. 'child\_of' => $ancestor,
17. 'depth' => $instance[ 'depth' ],
18. 'title\_li' => '',
19. );
21. // set a value for get\_pages to check if it's empty
22. $list\_pages = get\_pages( $args );
24. // check if $list\_pages has values
25. **if**( $list\_pages ) {
27. // open a list with the ancestor page at the top
28. ?>
29. <ul **class**="page-tree">
30. <?php // list ancestor page ?>
31. <li **class**="ancestor">
32. <a href="<?= get\_permalink( $ancestor ); ?>"><?= get\_the\_title( $ancestor ); ?></a>
33. </li>
35. <?php
36. // use wp\_list\_pages to list subpages of ancestor or current page
37. wp\_list\_pages( $args );

40. // close the page-tree list
41. ?>
42. </ul>
44. <?php
45. }
46. }
47. }

This checks if we're on a page and then defines the arguments for the $**list\_pages** variable using the output of the previous function and the value of the **$depth** variable which is set by the widget's form.

# SCRIPTS AND STYLES

We can add our own styling or JavaScript functionality by enqueueing the asset. Enqueuing uses actions to add scripts and styles modularly, taking care of any dependencies in the process.

For example, let’s add support for a Google Font, which is actually a stylesheet:

1. add\_action( 'wp\_enqueue\_scripts', 'my\_enqueued\_assets' );
3. **function** my\_enqueued\_assets() {
4. wp\_enqueue\_style( 'my-font', '//fonts.googleapis.com/css?family=Barrio' );
5. }

The action we use is horribly named because **wp\_enqueue\_scripts** is actually used to add both scripts and styles to the front-end. In the hooked function we use **wp\_enqueue\_style()** to add our style. The first parameter is the slug or handle of the script and the second parameter is the URL of the script.

Using **//url.com** instead of **http://url.com** is a neat trick which allows browsers to grab the appropriate version of the script. If our connection uses https it will retrieve the HTTPS version, otherwise it will use the regular HTTP version.

We also load assets we’ve made and store within our plugin. Let’s load a custom script we’ve made using the enqueueing method:

1. add\_action( 'wp\_enqueue\_scripts', 'my\_enqueued\_assets' );
3. **function** my\_enqueued\_assets() {
4. wp\_enqueue\_script( 'my-script', plugin\_dir\_url( **\_\_FILE\_\_** ) . '/js/my-script.js', **array**( 'jquery' ), '1.0', true );
5. }

The process is much the same but we’ve used more parameters in the **wp\_enqueue\_script()** function. The third parameter defines the dependencies of the script. WordPress makes sure to load all the dependencies properly so even if we enqueue a dependency later they will be loaded correctly. The fourth parameter is a version number we can choose ourself. These additional parameters are available for the **wp\_enqueue\_style()** function as well.

The fifth parameter, when set to **true**, instructs WordPress to load a script in the footer instead of the header. This is preferred if our scripts aren’t needed in the header specifically as it decreases loading times and optimizes JavaScript.