# Plugin Integration

Integrating a plugin to WordPress is essential for users to interact with it. WordPress features many areas where plugins can be integrated, including a meta box, dashboard widgets, and custom shortcodes.

## Creating a Meta Box

WordPress features multiple meta boxes on the Add New Post and Page screens. These meta boxes are used for adding additional information to posts, pages, and content.

Meta boxes can be created in a plugin using the add\_meta\_box() function in WordPress. This function accepts seven parameters, as shown here:

EXAMPLE:

<?php add\_meta\_box( $id, $title, $callback, $page, $context, $priority, $callback\_args ); ?>

SEE: <http://codex.wordpress.org/Function_Reference/add_meta_box>

Each parameter helps define where and how your meta box is displayed:

$id ([*string*](http://codex.wordpress.org/How_to_Pass_Tag_Parameters#String)) (*required*) HTML 'id' attribute of the edit screen section

Default: *None*

$title ([*string*](http://codex.wordpress.org/How_to_Pass_Tag_Parameters#String)) (*required*) Title of the edit screen section, visible to user

Default: *None*

$callback ([*callback*](http://codex.wordpress.org/How_to_Pass_Tag_Parameters#Callback)) (*required*) Function that prints out the HTML for the edit screen section. The function name as a string, or, within a class, an array to call one of the class's methods. The callback can accept up to two arguments: the first argument is the $post object for the post or page that is currently being edited. The second argument is the full $metabox item (an array), see [Callback args](http://codex.wordpress.org/Function_Reference/add_meta_box#Callback_args). See the second example under [Examples](http://codex.wordpress.org/Function_Reference/add_meta_box#Examples) below.

Default: *None*

$post\_type ([*string*](http://codex.wordpress.org/How_to_Pass_Tag_Parameters#String)) (*required*) The type of Write screen on which to show the edit screen section ('post', 'page', 'dashboard', 'link', 'attachment' or 'custom\_post\_type' where custom\_post\_type is the custom post type slug)

Default: *None*

$context ([*string*](http://codex.wordpress.org/How_to_Pass_Tag_Parameters#String)) (*optional*) The part of the page where the edit screen section should be shown ('normal', 'advanced', or 'side'). (Note that 'side' doesn't exist before 2.7)

Default: 'advanced'

$priority ([*string*](http://codex.wordpress.org/How_to_Pass_Tag_Parameters#String)) (*optional*) The priority within the context where the boxes should show ('high', 'core', 'default' or 'low')

Default: 'default'

$callback\_args ([*array*](http://codex.wordpress.org/How_to_Pass_Tag_Parameters#Array)) (*optional*) Arguments to pass into your callback function. The callback will receive the $post object and whatever parameters are passed through this variable.

Default: null

EXAMPLE:

<?php

add\_action ( ‘add\_meta\_boxes’, ‘myplugin\_meta\_box\_init’ );

//meta box functions for adding the meta box and saving the data

function myplugin\_meta\_box\_init() {

//create the custom meta box

add\_meta\_box( ‘myplugin-meta’, ‘Information’, ‘myplugin\_meta\_box’, ‘post’, ‘slide’, ‘default’ );

}

?>

The first step to adding a meta box is to use the add\_meta\_boxes action hook to execute the custom function myplugin\_meta\_box\_init() function. In this function, you call the add\_meta\_box() function to create your custom meta box for “Information.”

Set the CSS id attribute to myplugin-meta for your beta box. The second parameter is the title, which you set to Information. The next parameter is your custom function myplugin\_meta\_box(), which will display the HTML for the meta box. Next you define the meta box to display on the post page and in the sidebar. Finally, set the priority to default.

Now create the myplugin\_meta\_box() function to display the meta box fields.

EXAMPLE:

function myplugin\_meta\_box( $post, $box ) {

// retrieve the custom meta box values

$myplugin\_featured = get\_post\_meta( $post->ID, '\_myplugin\_type', true );

$myplugin\_price = get\_post\_meta( $post->ID, '\_myplugin\_price', true );

//nonce for security

wp\_nonce\_field( plugin\_basename( \_\_FILE\_\_ ), 'myplugin\_save\_meta\_box' );

// custom meta box form elements

echo '

<p>

Price: <input type="text" name="myplugin\_price"

value="'.esc\_attr( $myplugin\_price ).'" size="5" />

</p>

';

echo '

<p>Type:

<select name="myplugin\_product\_type" id="myplugin\_product\_type">

<option value="0" ' .selected( $myplugin\_featured, 'normal', false ). '>

Normal</option>

<option value="special" ' .selected( $myplugin\_featured, 'special', false ). '>

Special</option>

<option value="featured" ' .selected( $myplugin\_featured, 'featured', false ). '>

Featured</option>

<option value="clearance" ' .selected( $myplugin\_featured, 'clearance', false ). '>

Clearance</option>

</select>

</p>

';

}

The first step in your custom function is to retrieve the saved values for your meta box. If it is a new post, there will not be any saved values. Next display the form elements in the meta box. Notice there are no form tags or submit button. Also notice that the wp\_nonce\_field() function is active to create a custom nonce field in the form.

Now that the form elements are in place, the data needs to be saved when the post is saved. Another function will handle that, myplugin\_save\_meta\_box(). Trigger it by the save\_post action hook.

EXAMPLE:

// hook to save our meta box data when the post is saved

add\_action( 'save\_post', 'myplugin\_save\_meta\_box' );

function myplugin\_save\_meta\_box( $post\_id ) {

// process form data if $\_POST is set

if( isset( $\_POST['myplugin\_product\_type'] ) ) {

// if auto saving skip saving our meta box data

if ( defined( 'DOING\_AUTOSAVE' ) && DOING\_AUTOSAVE )

return;

//check nonce for security

check\_admin\_referer( plugin\_basename( \_\_FILE\_\_ ), 'myplugin\_save\_meta\_box' );

// save the meta box data as post meta using the post ID as a unique prefix

update\_post\_meta( $post\_id, '\_myplugin\_type', sanitize\_text\_field( $\_POST['myplugin\_product\_type'] ) );

update\_post\_meta( $post\_id, '\_myplugin\_price', sanitize\_text\_field( $\_POST['myplugin\_price'] ) );

}

}

The save\_post action hook runs whenever a post is saved in WordPress. Because it’s only needed with the custom meta data in the meta box, the first thing to do is verify that the $\_POST[‘myplugin\_product\_type’] value is set. Next, verify that the post being saved is an active post and not an auto save. Do this by checking that the post is not auto-saving and, if so, you exit the function. The next step is to verify that the nonce value is the expected value. If the post is active and your form elements have been set, it saves the form data. Once all checks have passed, use update\_post\_meta() to save the meta box data as metadata against your post.

Send in the post ID as the first parameter to update\_post\_meta(). This tells WordPress what post the meta data will be attached to. Next, pass in the name of the meta key that’s being updated. Notice the meta key name is prefixed with an underscore. This prevents these values from being listed in the custom fields meta box on the post edit screen. The final parameter sent is the new value for the meta key, which is being sanitized using the sanitize\_text\_field() WordPress function.

To display the data use the get\_post\_meta function inside the loop.

EXAMPLE:

<?php

$myplugin\_type - get\_post\_meta( $post->ID, ‘\_myplugin\_type’, true );

$myplugin\_price - get\_post\_meta( $post->ID, ‘\_myplugin\_price’, true );

echo ‘<p>Price: ‘ .esc\_html( $myplugin\_price ). ‘</p>’;

echo ‘<p>Type: ‘ .esc\_html( $myplugin\_type ). ‘</p>’;

?>

## Shortcodes

WordPress features a shortcode API that can be used to easily create shortcode functionality in plugins. Shortcodes are basically text macro codes that can be inserted into a post, page, or custom post type. When being displayed, those shortcodes are replaced by some other type of content.

EXAMPLE:

<?php

add\_shortcode( ‘mytwitter’, ‘myplugin\_twitter’ );

function myplugin\_twitter() {

return ‘<a href=”https://twitter.com/BenGoliwas>@BenGoliwas</a>’;

}

?>

Now, anytime [mytwitter] is put into page or post content, it will be replaced with an HTML link to Twitter.

Shortcodes can be configured to accept attributes. This is useful for passing arguments to your custom functions, thereby altering the output of the shortcode based on those arguments.

EXAMPLE:

<?php

add\_shortcode( ‘mytwitter’, ‘myplugin\_twitter’ );

function myplugin\_twitter( $atts, $content = null ) {

extract( shortcode\_atts( array(

‘person’ => ‘ben’ //set attribute default

), $atts ) );

if ( $person == ‘ben’ ) {

return ‘<a href=”<https://twitter.com/BenGoliwas>”>@BenGoliwas</a>’;

}elseif ( $person == ‘al’ ) {

return ‘<a href=”<https://twitter.com/algore>”>@AlGore</a>’;

}elseif ( $person == ‘dalai’ ) {

return ‘<a href=”<https://twitter.com/DalaiLama>”>@Dalai Lama</a>’;

}

}

?>

This code crates the same shortcode as before, but now an attribute called person is defined. With this attribute, a person can be specified. To display the Twitter URL for Al Gore, use the shortcode [mytwitter person=”al”]. Shortcodes can accept multiple attributes from the array set in the shortcode function.

## Creating a Widget

Widgets are a common feature included in many WordPress plugins. Creating a widget with a plugin easily gives users a way to add plugin information to the sidebar or other widgetized areas.

Widgets work in the WP\_Widget class in WordPress. This class features built-in functions for building for building a widget, each with a specific purpose.

SEE: <http://codex.wordpress.org/Widgets_API>

EXAMPLE:

<?php

class My\_Widget extends WP\_Widget {

function My\_Widget() {

// process the widget

}

function form($instance) {

//widget form in admin dashboard

}

function update($new\_instance, $old\_instance) {

//save widget options

}

function widget ($args, $instance) {

// display the widget

}

}

?>

From the WordPress Codex:

In technical terms: a WordPress Widget is a [PHP object](http://php.net/manual/en/language.oop5.php) that echoes string data to [STDOUT](http://en.wikipedia.org/wiki/Stdout#Standard_output_.28stdout.29) when its widget() method is called. It's located in [wp-includes/widgets.php](https://core.trac.wordpress.org/browser/tags/4.0/src/wp-includes/widgets.php#L0).

EXAMPLE:

add\_action( ‘widgets\_init’, ‘myplugin\_register\_widgets’ );

function myplugin\_register\_widgets() {

register\_widget( ‘myplugin\_widget’ );

}

Calling the action hook widgets\_init executes the function myplugin\_register\_widgets(), as shown in the preceding code. Here the widget is called myplugin\_widget.

The Widget API makes creating a widget easy. To begin, extend the pre-existing WP\_Widget class by creating a new class with a unique name. Then add the first function, which should be the same name as your unique class name.

EXAMPLE:

class myplugin\_widget extends WP\_Widget {

function myplugin\_widget() {

$widget\_ops = array(

‘classname’ => ‘myplugin\_widget\_class’,

‘description’ => ‘Example widget that displays a user\’s bio.’ );

$this ->WP\_Widget( ‘myplugin\_widget’, ‘Bio Widget’, $widget\_ops );

}

In this function you define the classname for the widget. The classname is the CSS class that will be added to the HTML tag wrapping the widget when it’s displayed. Depending on the theme the CSS class may be in a <div>, <aside>, <li>, or other HTML tag. You also set the description for your widget. This is displayed on the widget dashboard below the widget name. These options are then passed to WP\_Widget. It also passes the CSS ID name (myplugin\_widget\_class) and the widget name (Bio Widget).

Next create the function to build the widget settings form. Widget settings are located on the widget admin page upon expanding any widget listed on a sidebar.

//build our widget settings form

function form( $instance ) {

$defaults = array(

'title' => 'My Bio',

'name' => 'Ben Goliwas',

'bio' => ''

);

$instance = wp\_parse\_args( (array) $instance, $defaults );

$title = $instance['title'];

$name = $instance['name'];

$bio = $instance['bio'];

?>

<p>Title:

<input class="widefat" name="<?php echo $this->get\_field\_name( 'title' ); ?>" type="text" value="<?php echo esc\_attr( $title ); ?>" /></p>

<p>Name:

<input class="widefat" name="<?php echo $this->get\_field\_name( 'name' ); ?>" type="text" value="<?php echo esc\_attr( $name ); ?>" /></p>

<p>Bio:

<textarea class="widefat" name="<?php echo $this->get\_field\_name( 'bio' ); ?>"><?php echo esc\_textarea( $bio ); ?></textarea></p>

<?php

}

The first thing to do is define the default widget values. If the user doesn’t fill in the settings, you can default these values to whatever you like. In this case, you’re setting the default title to My Bio and default name to Ben Goliwas. Next pull in the instance values, which are your widget settings. If the widget was just added to a sidebar, there are no settings saved so these values will be empty. Finally, display the three form fields for the widget settings: title, name, and bio. The first two values are useing text input boxes and the bio value is using a textarea box.

Notice that you don’t need <form> tags or a submit button; the widget class will handle this. Remember to use the appropriate escaping functions when displaying your data, in this case esc\_attr() for the two text fields and esc\_textarea() for the textarea fields.

Next save the widget settings using the update() widget class function:

EXAMPLE:

//save our widget settings

function update( $new\_instance, $old\_instance ) {

$instance = $old\_instance;

$instance['title'] = sanitize\_text\_field( $new\_instance['title'] );

$instance['name'] = sanitize\_text\_field( $new\_instance['name'] );

$instance['bio'] = sanitize\_text\_field( $new\_instance['bio'] );

return $instance;

}

Notice that you don’t need to save the settings yourself, the widget class does it for you. Pass in the $new\_instance values for each of your setting fields. This strips out any HTML that might be entered with sanitize\_text\_field(). To accept HTML values, use wp\_kses() instead.

SEE:

<http://codex.wordpress.org/Function_Reference/sanitize_text_field>

&

<http://codex.wordpress.org/Function_Reference/wp_kses>

The final function in myplugin\_widget class handles displaying your widget:

//display our widget

function widget( $args, $instance ) {

extract( $args );

echo $before\_widget;

$title = apply\_filters( 'widget\_title', $instance['title'] );

$name = ( empty( $instance['name'] ) ) ? '&nbsp;' : $instance['name'];

$bio = ( empty( $instance['bio'] ) ) ? '&nbsp;' : $instance['bio'];

if ( !empty( $title ) ) { echo $before\_title . esc\_html( $title ) . $after\_title; };

echo '<p>Name: ' . esc\_html( $name ) . '</p>';

echo '<p>Bio: ' . esc\_html( $bio ) . '</p>';

echo $after\_widget;

}

First extract the $args parameter. This variable stores some global theme values such as $before\_widget and $after\_widget. These variables can be used by theme developers to customize what code will wrap your widget - for example, a custom <div> tag. After extracting the $args parameter, you display the $before\_widget variable. The $before\_title and $after\_title are also set in this variable. This is useful for passing custom HTML tags to wrap the widget title in.

Next, display widget values. The title is displayed first and wrapped by $before\_title and $after\_title.

## Create a Dashboard Widget

Dashboard widgets are displayed on the main dashboard of your WordPress installation.

SEE: <http://codex.wordpress.org/Dashboard_Widgets_API>

To create a custom Dashboard Widget, use the wp\_add\_dashboard\_widget() function.

EXAMPLE:

<?php

add\_action( ‘wp\_dashboard\_setup’, ‘myplugin\_add\_dashboard\_widget’ );

//call function to create our dashboard widget

function myplugin\_add\_dashboard\_widget() {

wp\_add\_dashboard\_widget( ‘myplugin\_dashboard\_widget’,

‘My Plugin Dashboard Widget’, ‘myplugin\_create\_dashboard\_widget’ );

}

// function to display our dashboard widget content

function myplugin\_create\_dashboard\_widget() {

echo ‘<p>Hello World! This is my Dashboard Widget</p>’;

}

?>

First call the wp\_dashboard\_setup action hook to execute the function to build your custom Dashboard Widget. This hook is triggered after all of the default Dashboard Widgets have been built. Next execute the wp\_add\_dashboard\_widget function to create your Dashboard Widget. The first parameter is the widget ID slug. This is used for the CSS classname and the key in the array of widgets. The next parameter is the display name for your widget. The final parameter sent is the custom function name to display the widget contents. An optional fourth parameter can be sent for a control callback function. This function would be used to process any form elements that might exist in the dashboard widget.

After executing the wp\_add\_dashboard\_widget() function the custom function is called to display the widget contents. In this example, it displays a simple string. The result is a custom dashboard widget.

SEE:

<http://codex.wordpress.org/Plugin_API/Action_Reference/wp_dashboard_setup>

&

<http://codex.wordpress.org/Function_Reference/wp_add_dashboard_widget>