Plugin Settings

Settings help users optimize the plugin to increase personalization and customization.

## Saving Plugin Options

WordPress features some very easy-to-use functions to save, edit, and delete options.

To create options there are two features: add\_option() and update\_option().

EXAMPLE:

<?php add\_option( ‘myplugin\_display\_mode’, ‘Option Name’ ); ?>

The first parameter you send to the add\_option() function is the name of your option. This is a required field and must be unique from all other options saved in WordPress, including from other plugins. The second parameter is the option value. This is also requires and can be a string, an array, an object, or a serialized value.

You can also use update\_option() to create new options. This function checks to see if the option exists. If it does not, it creates it. update\_option() is called exactly as the add\_option()

EXAMPLE:

<?php update\_option( ‘myplugin\_display\_mode’, ‘Option Name’ ); ?>

Generally it’s best practice to use the update\_option() for both adding and editing options.

To retrieving an option value, use the get\_option() function.

EXAMPLE:

<?php get\_option(‘myplugin\_display\_mode’ ); ?>

The only required field for get\_option() is the name of the option you want to retrieve. If the option exists, it is returned to display or it is stored in a variable. If it does not exist, the function returns FALSE.

Options can be deleted by using the delte\_option() function.

EXAMPLE:

<?php delete\_option( ‘myplugin\_display\_mode’ ); ?>

NOTE: It’s best practice to start all of your option names with the same prefix, “myplugin\_” is an example of this. This is useful in terms of creating uniqueness and readability.

Options in WordPress are not reserved for just plugins. Themes can also create options to store specific theme data.

## Array of Options

Every option created in WordPress adds a new record to the qp\_options database table. It’s a good idea to store these options in an array. This reduces the number of records in the database and the number of update\_options() calls you need to make.

EXAMPLE:

<?php

$myplugin\_options\_arr = array(

‘myplugin\_display\_mode’ => ‘Option Mode’,

‘myplugin\_default\_browser’ => ‘Chrome’,

‘myplugin\_favorite\_book’ => ‘Professional WordPress’,

);

update\_option(‘myplugin\_plugin\_options’, $myplugin\_options\_arr );

?>

In the above code you are creating an array to store your plugin option values. Rather than call update\_option() three times, and save three records in the database, you need to call it only once and save your array to the option named myplugin\_plugin\_options.

To retrieve the array of options, you use the same get\_option() function as before:

<?php

$myplugin\_options\_arr = get\_option( ‘myplugin\_plugin\_options’ );

$myplugin\_display\_mode = $myplugin\_options\_arr[‘myplugin\_display\_mode’];

$myplugin\_default\_browser = $myplugin\_options\_arr[‘myplugin\_default\_browser’];

$myplugin\_favorite\_book = $myplugin\_options\_arr[‘myplugin\_favorite\_book‘];

?>

## Creating a Menu and Submenus

WordPress features two different ways to create a custom menu for your plugin. Start with deciding on a location for your options page. It can be in it’s own top level menu or as a sub-menu of an existing menu.

## Creating a Top-Level Menu

Using a top-level menu is useful if your plugin has multiple settings pages that need to be separate. To create your own top-level menu, you’ll use ht eadd\_menu\_page() function.

EXAMPLE:

<?php add\_menu\_page( page\_title, menu\_title, capability, menu\_slug, function, icon\_url, position ); ?>

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

Create submenu items for your new menu using the add\_submenu\_page() function.

<?php add\_submenu\_page ( parent, page\_title, menu\_title, capability, menu\_slug, [function] );

?>

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

NOTE: If you're running into the "You do not have sufficient permissions to access this page." message in a wp\_die() screen, then you've hooked too early. The hook you should use is admin\_menu.

<?php

//create custom plugin settings menu

add\_action( ‘admin\_menu’, ‘myplugin\_create\_menu’ );

function myplugin\_create\_menu() {

//create new top-level menu

add\_menu\_page (‘MyPlugin Settings’ , ‘Myplugin’,

‘manage\_options’, ‘myplugin\_main\_menu’, ‘myplugin\_main\_plugin\_page’,

plugins\_url ( ‘/images/wordpress.png’, \_\_FILE\_\_) );

//create two sub-menus: settings and support

add\_submenu\_page( ‘myplugin\_main\_menu’, ‘Myplugin Settings’,

‘Settings’, ‘manage\_options’, ‘myplugin\_settings’, myplugin\_settings\_page’ );

add\_submenu\_page( ‘myplugin\_main\_menu’, ‘Myplugin Settings’,

‘Support’, ‘manage\_options’, ‘Myplugin\_settings’, ‘myplugin\_support\_page’ );

}

First you call the admin\_menu action hook. This is triggered after the basic admin panel menu structure is in place. Once triggered, you call your custom function myplugin\_create\_menu() to build your menu.

To create your menu, you call the add\_menu\_page() function. THe first two parameters set your page title and menu title. You also set the capability level to manage\_options so only an admin will see this new menu. Next, you set the menu slug to myplugin\_main\_menu, which is the unique slug for your menu. Your custom menu function name is next, in this case myplugin\_main\_plugin\_page. Remember that you haven’t created this function yet so when viewing the settings page, you will get a PHP warning. Finally, set the custom icon location to display the WordPress logo.

Now that the top-level menu is created, create the submenu items. In the above example you are creating two submenu items: Settings and Support. To do this, you use the add\_submenu\_page() function.

The first parameter you send is the main slug of the top-level menu you want this to fall under. You set this to myplugin\_main)menu, which is a unique slug for your plugin menu. Next, set the page title and menu title just like the primary menu item. Also the access level for viewing is set with manage\_options. Each submenu item must have a unique menu slug. The final value is the custom function to build the settings page for each submenu.

## Adding to an Existing Menu

Most plugins have only one options page and therefore do not require an entirely separate top-level menu. To add a settings page to an existing menu, it’s possible to add a plugin option page to any existing menu in WordPress.

EXAMPLE:

<?php

add\_action( ‘admin\_menu’, ‘myplugin\_create\_settings\_submenu’ );

function myplugin\_create\_settings\_submenu() {

add\_options\_page( ‘Myplugin Settings Page’, ‘Myplugin Settings’,

‘manage\_options’, ‘myplugin\_settings\_menu’, ‘myplugin\_settings\_page’ );

}

?>

The first parameter is the page title followed by the submenu display name. Set the capability to manage\_options, so the menu is viewable only by administrators. Next, set the unique menu handle to myplugin\_settings\_menu. Then, call the custom myplugin\_settings\_page() function to build your options page. This example will place the link at the bottom of the settings menu.

SEE: <http://codex.wordpress.org/Function_Reference/add_submenu_page#Parameters>

## Creating an Options Page

WordPress settings API is a powerful set of functions to help make saving options in WordPress easy and secure. One of the major benefits of the settings API is that WordPress handles the security checks, meaning you don’t need to include a nonce in your form.

In the add\_menu\_page() and add\_submenu\_page() functions the menu item is defined to display your options page. To create an options page, you need to create this function to display your options.

EXAMPLE:

<?php

//create custom plugin settings menu

add\_action( ‘admin\_menu’, ‘myplugin\_create\_menu’ );

function myplugin\_create\_menu() {

//create new top-level menu

add\_menu\_page(‘My Plugin Page, ‘Halloween Plugin’,

‘manage\_options’, ‘myplugin\_main\_menu’, ‘myplugin\_setings\_page’,

plugins\_url( ‘/images/wordpress.png’, \_\_FILE\_\_) );

//call register settings function

add\_action( ‘admin\_init’, ‘myplugin\_register\_settings’ );

}

}

?>

Notice the new Action Hook for admin \_init to execute your myplugin\_register\_settings() function, as shown in the following code:

EXAMPLE

<?php

function myplugin\_register\_settings() {

//register our settings

register\_settings( ‘myplugin-settings-group’, ‘myplugin\_options’,

‘myplugin\_sanitize\_options’);

}

?>

Using the settings API’s register\_setting() function, you define the option you are going to offer on your plugin options page. The settings page will have three options, but you are going to store those three options in a single options array, so there is only one setting to register here. The first parameter is the options group name. This required field needs to be a group name to identify all option sin this set. The second parameter is the actual option name and must be unique. The third is a callback function to sanitize the options value. Next, build the options page. Do this by creating the myplugin\_settings\_page() function as called from your menu:

EXAMPLE:

<?php

function myplugin\_settings\_page() {

?>

<div class=”wrap”>

<h1>My Plugin Options</h1>

<form method=”post” action=”options.php”>

<?php settings\_fields( ‘myplugin-settings-group’ ); ?>

<?php $myplugin\_options = get\_option( ‘myplugin\_options’ ); ?>

<table class="form-table">

<tr valign="top">

<th scope="row">Name</th>

<td><input type="text" name="myplugin\_options[option\_name]" value="<?php echo esc\_attr( $myplugin\_options['option\_name'] ); ?>" /></td>

</tr>

<tr valign="top">

<th scope="row">Email</th>

<td><input type="text" name="myplugin\_options[option\_email]" value="<?php echo esc\_attr( $myplugin\_options['option\_email'] ); ?>" /></td>

</tr>

<tr valign="top">

<th scope="row">URL</th>

<td><input type="text" name="myplugin\_options[option\_url]" value="<?php echo esc\_url( $myplugin\_options['option\_url'] ); ?>" /></td>

</tr>

</table>

<p class="submit">

<input type="submit" class="button-primary" value="Save Changes" />

</p>

</form>

</div>

<?php

}

?>

Some noticeable differences between this and a standard form. The <form> tag must be set to post to options.php. The settings group (myplugin-settings-group) must be defined. This establishes the link between your options and their values. This is done with:

EXAMPLE:

<?php settings\_fields( ‘myplugin-settings-group’ ); ?>

Next, load the existing options array, if there are any, to the $myplugin\_options variable using the get\_option() function. This is used to display the existing options that are set in the form.

Then the table displaying the form is built. Notice the name of the form field needs to be in the format of option\_name[field\_name]. This stores all option values in a single array.

EXAMPLE

<input type="text" name="myplugin\_options[option\_email]" value="

<?php echo esc\_attr( $myplugin\_options['option\_email'] ); ?>" />

After all forms display a submit button to post the form and save the options.

The final step is to create the myplugin\_sanitize\_options() function. This will be used to sanitize all data submitted in your plugin settings prior to saving in the database. This is extremely important because unsanitized data could potentially open up a security vulnerability in your plugin.

EXAMPLE:

<?php

function myplugin\_sanitize\_options( $input ) {

$input['option\_name'] = sanitize\_text\_field( $input['option\_name'] );

$input['option\_email'] = sanitize\_email( $input['option\_email'] );

$input['option\_url'] = esc\_url( $input['option\_url'] );

return $input;

}

?>

Notice how each option value is being sanitized with a specific function. The name option uses the WordPress function sanitize\_text\_field() to strip any HTML, XML, and PHP tags from the submitted value. Use the sanitize\_email() WordPress function to sanitize the email value and esc\_url() to sanitize the URL value.

That’s it!