WordPress Plugin Template

Information obtained from http://plugin.michael-simpson.com/

Design Tenets

The aim of this template code is to provide a code pattern for managing complexity of a WordPress plugin and following best practices. High-level issues are the following.

Life Cycle Management
By default, a plugin can't tell if it's being installed, been installed already, or even being upgraded. Maybe you want to create a DB table during installation and later you add a column to the table when the user upgrades to version 1.6. But you don't want the plugin to try these things every time it is accessed or activated.

Your XXX_Plugin class inherits from a class called XXX_PluginLifeCycle. This class has functions for installation, activation, deactivation, upgrade, uninstall and more. Overriding them in your class gives you a hook to write code to happen during these events. Also included is a check for your minimally-required PHP version.

Easier Options Handling
Practically every plugin has options (a.k.a "settings") for users to set. The plugin template code provides a quick way to define a set up options and their values. An options page for the plugin can be automatically installed in the administration menus. Options are automatically deleted on uninstall. Options that define roles (like what role is minimally required to perform a function) are easy to define and enforce.

Prefixing Options
Part of being a good plugin citizen is avoiding having the same name for an option as any other plugin. Otherwise, they collide in the wp_options table. This plugin provides alternative to get_option, add_option, update_option, delete_option functions:
getOption, addOption, updateOption, deleteOption which will automatically add a prefix to the option name stored in the DB to avoid collisions. getOption also allows you to return a default value if the option is not set.

Short Code Support
Support for creating and loading multiple short codes. Includes a technique to avoid loading scripts needed by a short code into pages where the short code does not appear (a best practice)

More Easy to Understand Roles for Security
Provides easier role names to work with
(Administrator, Editor, Author, Contributor, Subscriber, Anyone) with easy way to map to WP "capability". Convenience functions like isUserRoleEqualOrBetterThan($roleName) make programmatic security checks easy. Options include a notion of "role options" where an option can be set with a choice of roles. That can be used by an administrator to set role security on a plugin's operations (the ones you write).

Support for Internationalization
Template code includes internationalized strings and the plugin initialization automatically gets set up to load your text domain and translation files.

Convenience Functions
Convenience functions are provided for common operations

Object-Oriented Design
Use of Object-Oriented Design to support code re-use and abstraction through inheritance (This means it requires PHP 5 or later).

Getting Started

After you **download** your generated code, get oriented by looking at the **Files Overview**. You do most of your coding in the XXX_Plugin.php class file.
You may need certain operations to happen during **install/uninstall** (including **making database tables**), during **activation/deactivation**, and during **upgrades**.
You probably want to **add your own actions and filters** including **en-queuing Javascript & CSS Files**
Maybe **add short codes**
You may need to **add some AJAX handlers**
Your plugin may need to have **options** including an **options administration page**.
You will want to understand how to **enforce role security**.
You may need to **add administration pages**.
As a good plugin writer, you will want your plugin to support **internationalization.**
Once you have your plugin created, you may want to share it. To do this, **register your code on WP.org**.

You will later want to release and upgrade your plugin.

Download Template Plugin Code

This page provides a form to download template code for writing a WordPress plugin. However, you don't need any of this to write a WordPress plugin. Especially a simple one. You can get along OK by following the standard directions on how to create a plugin.

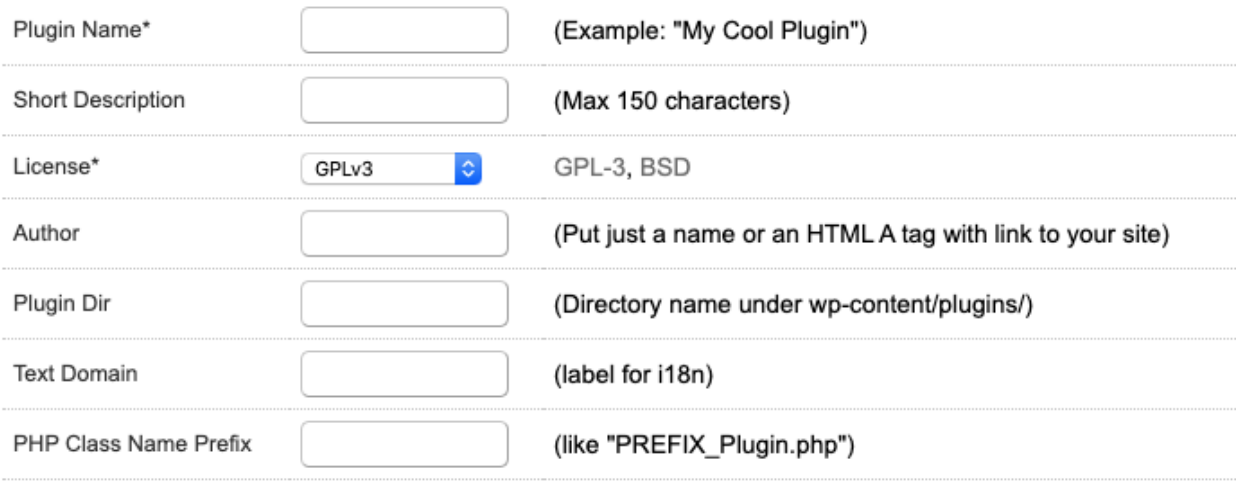
So why bother? If you write a more complex plugin, or several plugins, you have to figure out how to manage things like actions that should only happen on the first install, actions to take when upgrading from version X to version Y, and creating some boiler plate code related to creating an administrative options page, short codes and more. You also want to exercise best practices. See more on the Design Tenets page.

This template code and supporting documentation is my cut at template code that provides a code pattern for managing complexity, boiler plate code, convenience functions, and best practices for being a "good plugin citizen."

Generate code for your plugin and see the Getting Started page.

NOTICE: You must go to Michael Simpson's website in order to use this form. Go to this link and then click on the "Download Template Plugin Code" link on the right.

NOTE: this template does NOT include support for WordPress Multisite.



Files Overview

Assuming you create “My Cool Plugin” you would have the following files and directories:

my-cool-plugin.php	The main plugin file. This one contains the plugin **standard header**. This file does a PHP version check and sets up the text domain.
my-cool-plugin_init.php	This file is called the the main file and initializes the plugin class defined in MyCoolPlugin_Plugin.php. It is separate so that is is not parsed unless PHP version check is successful (otherwise you would get a PHP parse error when trying to activate the plugin)
MyCoolPlugin_Plugin.php	Your plugin class. This is where your write your code. Extends MyCoolPlugin_LifeCycle.
MyCoolPlugin_LifeCycle.php	Superclass for MyCoolPlugin_Plugin. Provides lifecycle functions for actions related to installation, uninstall, activate, deactive, and upgrade. Override these functions in MyCoolPlugin_Plugin to add actions.
MyCoolPlugin_InstallIndicator.php	Superclass of MyCoolPlugin_LifeCycle. Includes functionality for tracking what version of your plugin is installed.
MyCoolPlugin_OptionsManager.php	Superclass of MyCoolPlugin_InstallIndicator. Provides management functions for options including creating an options page.
MyCoolPlugin_ShortCodeLoader.php	If you would like to create a short code in its own class, extend this class and implement one abstract function.
MyCoolPlugin_ShortCodeScriptLoader.php	Like MyCoolPlugin_ShortCodeLoader (a subclass of it) that includes another abstract method to implement aimed at including links to Javascript files in the footer only when a page uses the short code.
readme.txt	Standard plugin readme.txt file. **Example format.** **Format Validator**. **Mark-down syntax**.
css/	Empty directory in which to put style sheets files.
js/	Empty directory in which to place javascript files
languages/	Empty directory in which to put your i18n files (*.pot, *.po, *.mo)

Install/Uninstall Actions

You may want certain actions to happen during installation and uninstall. A typical example is the creation/deletion of a database table for the plugin.

Your XXX_Plugin class inherits install() and uninstall() functions from its superclass XXX_LifeCycle. ***DO NOT OVERRIDE*** these functions. Instead, look at the code. Each one calls other functions defined in XXX_LifeCycle that you can override in your XXX_Plugin class.
First, look at install:

```
    public function install() {

        // Initialize Plugin Options
        $this->initOptions();

        // Initialize DB Tables used by the plugin
        $this->installDatabaseTables();

        // Other Plugin initialization - for the plugin writer to override as needed
        $this->otherInstall();

        // Record the installed version
        $this->saveInstalledVersion();

        // To avoid running install() more then once
        $this->markAsInstalled();
    }

    public function uninstall() {
        $this->otherUninstall();
        $this->unInstallDatabaseTables();
        $this->deleteSavedOptions();
        $this->markAsUnInstalled();
    }
```

Override installDatabaseTables() and otherInstall() as needed.

```
    protected function unInstallDatabaseTables() {
    }
```

```
    protected function otherInstall() {
    }
```

Looking at uninstall:

```
    public function uninstall() {
        $this->otherUninstall();
        $this->unInstallDatabaseTables();
        $this->deleteSavedOptions();
        $this->markAsUnInstalled();
    }
```

Override otherUninstall and uninstallDatabaseTables() as needed

```
    protected function otherUninstall() {
    }
```

```
    protected function unInstallDatabaseTables() {
    }
```

Creating Database Tables During Install

Your plugin is activated each time a user accesses a page. WP does not give you a simple way to track if that activation means your plugin is installed for the first time, being upgraded, or just another page access. The template code from this site differentiates these component life cycle events and gives you a place to write code for them. One such important event is installation. By installation, this actually means the first activation after install. Read about the general mechanism of add **Install/Uninstall Actions**. That article describes different places to put code to be executed during installation. (Also see where to put code to execute during an update in **Update Actions**.

The most common thing to do during installation is to create a database table that your plugin will use. There is a specific function already for this purpose. All you need to do is override it in you XXX_Plugin class.

Locate this function which should be already in your XXX_Plugin class:

```
/**
 * See: http://plugin.michael-simpson.com/?page_id=101
 * Called by install() to create any database tables if needed.
 * Best Practice:
 * (1) Prefix all table names with $wpdb->prefix
 * (2) make table names lower case only
 * @return void
 */
protected function installDatabaseTables() {
    //        global $wpdb;
```

```
12      //          $tableName = $this->prefixTableName('mytable');
13      //          $wpdb->query("CREATE TABLE IF NOT EXISTS `$tableName`
14  (
        //              `id` INTEGER NOT NULL");
    }
```

To create a table, simply uncomment these three lines and change ‘mytable’ to the table name you wish to use.

Note: the “prefixTableName” function actually adds two prefixes to the table name. The function is:

```
    /**
     * @param  $name string name of a database table
     * @return string input prefixed with the WordPress DB table
1   prefix
2    * plus the prefix for this plugin (lower-cased) to avoid table
3   name collisions.
4    * The plugin prefix is lower-cases as a best practice that all DB
5   table names are lower case to
6    * avoid issues on some platforms
7    */
8   protected function prefixTableName($name) {
9       global $wpdb;
10      return $wpdb->prefix .  strtolower($this->prefix($name));
11  }
```

The $wpdb->prefix is a WP convention to prefix all tables with “wp_”. This prefix can be changed on a WP installation. This allows for more than one WP installation to share the same DB by simply using different table prefixes (and hence different tables).

This template code uses the convention that things like table names and options should be prefixed with the name of the plugin (actually the “XXX” in your “XXX_Plugin”) to avoid having the same names as other plugins. The use of $this->prefixTableName(), $this->prefix() and option handling functions like $this->getOption() are convenience functions to make this prefixing somewhat transparent.

If you use $this->prefixTableName(‘mytable’) in your MyAwesomePlugin_Plugin.php file, you would end up a tabled named “wp_myawesomeplugin_mytable”. Unlike option prefixes, table name prefixes are lower-cased because mixed case table names can cause a problem when you transfer a MySQL DB content between MySQL instances on Unix and Windows.

Dropping DB Tables on Uninstall

A good plugin citizen should clean up after itself. If your plugin creates DB tables on install, it makes sense to drop them on uninstall. To do this, override your unInstallDatabaseTable() function in your XXX_Plugin class and drop your tables there.

But it is not always that simple. Sometimes when a user has a problem with a plugin, he will try uninstalling it and installing it again. He may not intend for all his data to be removed.

One way I have dealt with that in the past is to create an option for the plugin where the plugin administration can choose if the tables should be dropped on uninstall. For example, I created a "DropOnUninstall" option like this (See **Handling Options**)

```
public function getOptionMetaData() {
    //  http://plugin.michael-simpson.com/?page_id=31
    return array(
        //'_version' => array('Installed Version'), // Leave this
one commented-out. Uncomment to test upgrades.
        //'DropOnUninstall' => array(__('Drop this plugin\'s
Database table on uninstall', 'TEXT_DOMAIN'), 'false', 'true')
    );
}
```

Then I added a check in the unInstallDatabaseTable() function like what is shown commented-out in the template code:

```
/**
 * See: http://plugin.michael-simpson.com/?page_id=101
 * Drop plugin-created tables on uninstall.
 * Issue: sometimes people have issues with a version of the
plugin and they will
 * uninstall with the intention of immediately re-installing. This
will cause loss of
 * data. Good practice is to set an option for whether or not to
drop tables on uninstall
 * @return void
 */
protected function unInstallDatabaseTables() {
    //        // if you use this 'DropOnUninstall' be sure to add
it to the array in getOptionMetaData()
    //          and you should also guard deleteSavedOptions()
    //        if ('true' === $this->getOption('DropOnUninstall',
'false')) {
    //            global $wpdb;
    //            $tableName = $this->prefixTableName('mytable');
    //            $wpdb->query("DROP TABLE IF EXISTS
`$tableName`");
    //        }
}
```

Actions on Activation & Deactivation

Plugins register activation and deactivation hooks. By default, the XXX_Plugin class functions activate() and deactivate are registered respectively. (This is set up in the XXX_main.php file)

To add code to the hooks, override these functions in your XXX_Plugin.php file and add your code

```
    public function activate() {
    }

    public function deactivate() {
    }
```

Upgrade Actions

Sometimes you may need to perform an action when your plugin is upgraded from version X to version Y. An example is to alter a database table that the plugin uses (such as add a new column).

Background: your plugin looks at two versions:

1. The version that is in the code of your main plugin file
2. The version that it secretly saves in the wp_options table

Your plugin tracks what version of your plugin is installed in and XXX_version option (where XXX is your plugin class name prefix). Your code indicates what version it is in its main plugin file (your-plugin-name.php). At the top is a special comment header that is used to give some metadata about your plugin. There is a line there like: “Version: 0.1”.
In plugin class file (XXX_Plugin.php), in the upgrade() function, you would write code to see if the version of the code saved in the DB is less than the version defined in the code. If so, perform your upgrade.
The code looks like this:

```
public function upgrade() {
    $upgradeOk = true;
    $savedVersion = $this->getVersionSaved();
    if ($this->isVersionLessThan($savedVersion, '2.0')) {
        if ($this->isVersionLessThan($savedVersion, '1.8')) {
            if ($this->isVersionLessThan($savedVersion, '1.5')) {
                // perform version 1.5 upgrade action
```

```
            }
            // perform version 1.8 upgrade action
        }
        // perform version 2.0 upgrade action
     }

    // Post-upgrade, set the current version in the options
    $codeVersion = $this->getVersion();
    if ($upgradeOk && $savedVersion != $codeVersion) {
        $this->saveInstalledVersion();
    }
}
```

The nesting of the IF's may seem a little strange at first. For performance we want to avoid checking all the version every time a page is loaded. Every time a page is loaded with the plugin activated, it needs to figure out if it is now activated as a new installed, upgrade, or just a page load. WordPress doesn't tell the plugin which, so it has to figure it out (happens in the XXX_init.php file). We want to make the overhead of that check minimal. So we put earlier upgrades inside the later upgrade checks. Then we perform upgrades at the end of an IF clause to ensure that upgrades proceed in order.

Imagine in this example, the user had version 1.1 installed then upgraded one time but to version 2.0. We want to ensure that the 1.5 upgrade action happens first, then the 1.8 upgrade, then 2.0. After that, on each page load, on the check for version <2.0 would happen (and fail since it is now 2.0)

At the end of a successful upgrade, you must save the code's version to the database. This is done by the "saveInstalledVersion" function. In the above example, we added a $upgradeOk variable and check it. This is not necessary, but you might want to check for errors during an upgrade (like can't do something in the DB because of permissions). If you then do not save the new version to the DB, the upgrade action will be tried again on each page load until it succeeds. You may or may not want this behavior.

Adding Filters & Actions

Inside your XXX_Plugin.php class file find the function addActionsAndFilters(). Make your calls to **add_action** and **add_filter** there.
The call signatures look like this:

add_action($tag, $function_to_add, $priority, $accepted_args);

add_filter($tag, $function_to_add, $priority, $accepted_args);

Create a function in your class to do the action or filter. Pass it to the add_action/add_filter using array(object, function-name) syntax. For example:

```
public function addActionsAndFilters() {
    add_action('action_hook', array(&$this, 'doMyAction'));
}

public function doMyAction() {
    // do it
}
```

Creating Short Codes

Short codes are introduced into WP via plugins. The template code provides you with some options to help manage complexity around short codes.

Option 1: you can create a short code that calls a function on your XXX_Plugin class
Option 2: you can create a short code in its own PHP class, by extending **XXX_ShortCodeLoader**. If you are going to write a lot of code & functions, it may be a better idea to separate that PHP code into another file rather than cram it all into the same XXX_Plugin.php file.
Option 3: like option 2, but with some support to only add its needed script and styles on pages where the plugin actually is called. This is done by extending **XXX_ShortCodeScriptLoader**.
Option 1: Short Code as Function on XXX_Plugin.php
To create a simple short code, find the addActionsAndFilters() and put in a call to add_shortcode that points to a function that you add.

```
public function addActionsAndFilters($atts) {
    add_shortcode('say-hello-world', array($this, 'doMyShortcode'));
}

public function doMyShortcode() {
    return 'Hello Word!';
}
```

Add **[say-hello-world]** to a post to invoke the short code. The $atts variable will be an array of the short code attributes. See add_shortcode.
NOTE: notice how the short code function returns its output string instead of echoing it. This is important. If you echo it, it will appear at the top of the page, not in the post where the short code is

inserted. Be sure to return output only. If you need to use echo, then buffer the output and return it using the following technique.

```
public function doMyShortcode() {
    ob_start();
    echo 'Hello World!';
    $output = ob_get_contents();
    ob_end_clean();
    return $output;
}
```

Option 2: Short Code as a Subclass of XXX_ShortCodeLoader

If your short code implementation involves a lot of code, or you have a lot of short codes, then you may find making a short code be its own class in its own file easier to manage. To do this, create a new file in your plugin directory. For this example, we create an XXX_HelloWorldShortCode.php file (remember “XXX” is the prefix used on the files for your plugin. This is just a convention to avoid class name collisions with other plugins).

```
include_once('XXX_ShortCodeLoader.php');

class XXX_HelloWorldShortCode extends XXX_ShortCodeLoader {
    /**
      * @param  $atts shortcode inputs
      * @return string shortcode content
      */
    public function handleShortcode($atts) {
        return 'Hello World!';
    }
}
```

Remember to return instead of echo output as described in the note under Option 1. The $atts variable will be an array of the short code attributes. See add_shortcode.

Register the short code in the XXX_Plugin.php file, addActionsAndFilters() function:

```
public function addActionsAndFilters() {
    include_once('XXX_HelloWorldShortCode.php');
       $sc = new XXX_HelloWorldShortCode();
       $sc->register('say-hello-world');
}
```

Option 3: Short Code as a Subclass of XXX_ShortCodeScriptLoader

This option works the same as Option 2 with two differences:

- Your new short code class extends XXX_ShortCodeScriptLoader instead of XXX_ShortCodeLoader
- You must implement an additional function in your class, addScript()

The XXX_ShortCodeScriptLoader super class helps you insert scripts only on those pages in which the short code appears. This is a somewhat imperfect solution. The problem is how WP processes pages. Ideally, we would like the short code to be able to inject scripts and styles in the HTML HEAD of the page in which it appears. But WP generates the HEAD before it ever gets to calling code associated with a short code. So any calls to wp_enqueue_script or wp_enqueue_style in your short code callback function do nothing. Your options are:

- En-queue scripts and style that the short code needs in all pages, even those in which the short codes does not appear. This is done in the XXX_Plugin::addActionsAndFiters() function. See **Enqueueing Javascript & CSS Files**
- Inject the scripts and styles in-line where the short code appears. The problem with this is if the short code appears more than once in a page, you include things multiple times.
- Put code in your short code callback function to include things in the footer

This XXX_ShortCodeScriptLoader employs the last option: including things in the footer. Here is an example:

```
include_once('XXX_ShortCodeScriptLoader.php');

class XXX_HelloWorldShortCode extends XXX_ShortCodeScriptLoader {

    static $addedAlready = false;
    public function handleShortcode($atts) {
        return 'Hello World!';
    }

    public function addScript() {
        if (!self::$addedAlready) {
            self::$addedAlready = true;
            wp_register_script('my-script', plugins_url('js/my-
script.js',
                    __FILE__), array('jquery'), '1.0', true);
            wp_print_scripts('my-script');
        }
    }
}
```

Notice the static $addedAlready variable used to guard against adding the script twice. There is some code in the super class that prevents the script from being added if the short code does not appear in the page.

This technique does not work for styles. Styles seem to need to be added before the HTML that they are styling. {Anybody know a trick to make this work?}

Register the short code just as in Option 2 above:

```
public function addActionsAndFilters() {
    include_once('XXX_HelloWorldShortCode.php');
       $sc = new XXX_HelloWorldShortCode();
       $sc->register('say-hello-world');
}
```

Enqueueing Javascript & CSS Files

Your plugin may require Javascript libraries and/or CSS. These are included using the wp_enqueue_script() and wp_enqueue_style().
But before you go en-queuing things, you should understand how to exercise some restraint. If your plugin simply en-queues a script or style, it will be included on EVERY page, including administrative and regular pages. Often you don't want that one every page. The more places it appears, the more likely it will cause a conflict with some other script or css from other plugins or from the theme. As a best practice, limit scripts to just those pages you need it. WP does not make this easy, but there are some simple tricks.

Generally, to en-queue something, in your XXX_Plugin.php file, find your addActionsAndFilters() function and add them there. There are some commented-out examples in the file you download. This example shows how to en-queue a script known to WP (jquery) and a style and script that you create for your plugin and place in your plugin's css/ and js/ sub-directories respectively.

Note: in an earlier version of this code, wp_enqueue_script() and wp_enqueue_style() functions were called directly in addActionsAndFilters(). But recent versions of WP (since 3.6?) give warnings about these calls when WP_DEBUG=true. Recent WP versions require that these functions be called on the wp_enqueue_scripts or admin_enqueue_scripts hooks. Now, we place the calls to those functions in a wrapper function, and in addActionsAndFilters() register the wrapper function to a hook using add_action as shown below.

```
public function addActionsAndFilters() {
    // enqueue scripts and styles for regular pages
    add_action('wp_enqueue_scripts', array(&$this,
'enqueueStylesAndScripts'));

    // enqueue scripts and styles for admin pages
    add_action('admin_enqueue_scripts', array(&$this,
'enqueueAdminPageStylesAndScripts'));
}

public function enqueueStylesAndScripts() {
    wp_enqueue_script('jquery');
    wp_enqueue_style('my-style', plugins_url('/css/my-style.css',
__FILE__));
    wp_enqueue_script('my-script', plugins_url('/js/my-
script.js', __FILE__));
}

public function enqueueAdminPageStylesAndScripts() {
    wp_enqueue_script('jquery');
    wp_enqueue_style('my-style', plugins_url('/css/my-style.css',
__FILE__));
    wp_enqueue_script('my-script', plugins_url('/js/my-
script.js', __FILE__));
}
```

The wp_enqueue_scripts adds these scripts & styles to regular pages while the admin_enqueue_scripts hooks adds them to all admin pages. There are a couple obvious examples of when you don't want that to happen:

1. A special administration page that you create that has its own special script or style.
2. A short code that needs script or style. You only want that added to pages where the short code appears

Regarding (1), The technique is to check the URL being requested to see if it is a page that requires the script or style. If you create your own administration page, when you register it you give it a "slug". A slug is part of the URL that identifies it. Create conditional code like this:

```
public function enqueueAdminPageStylesAndScripts() {
    // Apply scripts & styles to the plugin's admin setting page
only
    if (strpos($_SERVER['REQUEST_URI'], $this-
>getSettingsSlug()) !== false) {
        wp_enqueue_script('jquery');
        wp_enqueue_style('my-style', plugins_url('/css/my-
style.css', __FILE__));
        wp_enqueue_script('my-script', plugins_url('/js/my-
script.js', __FILE__));
    }
}
```

See more on the "Adding Administration Pages".

Regarding (2), there is a discussion of a technique supported by this template code for en-queuing only when the short code appears on the page. See the page on short codes for more information.

Creating Ajax Calls

AJAX handlers are easy to add. This involves two things: (1) a function to handle an AJAX call and (2) that method being registered to a URL.

See also: http://codex.wordpress.org/AJAX_in_Plugins

1. Create a function to handle an AJAX call

In your your XXX_Plugin.php class file, add a new function call like this:

```
public function ajaxACTION() {
    // Don't let IE cache this request
    header("Pragma: no-cache");
    header("Cache-Control: no-cache, must-revalidate");
    header("Expires: Thu, 01 Jan 1970 00:00:00 GMT");

    header("Content-type: text/plain");

    echo 'hello world';
    die();
}
```

Change "ACTION" in the function name to something meaningful to you.

The first three header lines ensure that browsers do not cache the results of the first call to this function via AJAX (after which it would simply return that cached results instead of actually making the call). IE tends to cache by default. This usually doesn't make sense for AJAX calls.

Change the content type header to what you want (like "application/json" for JSON).

The "echo 'Hello World'" line is where your content would go.

You need to end with the "die()" call in AJAX handler functions. This is related to how AJAX requests are handled by WordPress.

2. Register the function to an AJAX URL

In the same XXX_Plugin.php file, find the function addActionsAndFilters(). In that function register your AJAX function to a WP AJAX action name:

```
public function addActionsAndFilters() {
   add_action('wp_ajax_ACTION', array(&$this, 'ajaxACTION'));
   add_action('wp_ajax_nopriv_ACTION', array(&$this,
'ajaxACTION')); // optional
}
```

Where "**ajaxACTION**" is the name of your function. The "**ACTION**" part of "wp_ajax_ACTION" and "wp_ajax_nopriv_ACTION" is a unique name for your AJAX action.

- Registering wp_ajax_ACTION makes the AJAX call accessible to users who are logged into your site and have privileges. This is for security. You always need to have this line.
- Only add the wp_ajax_nopriv_ACTION line if you want the AJAX call to be accessible to non-logged-in users.
- NOTE: if you added the wp_ajax_nopriv_ACTION one but not the wp_ajax_ACTION one, then the AJAX call would be available to non-logged-in users but NOT to logged in users.

3. The AJAX URL

OK, you created a function and registered it. But what is the URL to it? The URL can be composed via

```
admin_url('admin-ajax.php') . '?action=' . $actionName;
```

for which there is a convenience function

```
$this->getAjaxUrl($actionName);
```

You can add additional GET parameters to an AJAX call like this:

```
$plainUrl = $this->getAjaxUrl('MyAjaxActionName);
$urlWithId = $this->getAjaxUrl('MyAjaxActionName&id=MyId);

// More sophisticated:
$parametrizedUrl = $this-
>getAjaxUrl('MyAjaxActionName&id=%s&lat=%s&lng=%s);
$urlWithParamsSet = sprintf($parametrizedUrl, urlencode($myId),
urlencode($myLat), urlencode($myLng));
```

4. Role-Based Security in AJAX Calls

wp_ajax_ACTION and wp_ajax_nopriv_ACTION are a bit coarse-grained. A more nuanced security check can be done programmatically in the function:

```
public function ajaxACTION() {
    if (!$this->isUserRoleEqualOrBetterThan('Author')) {
        die(1);
   }
    // do operation
    die();
}
```

Here we check that the user has at least Author role.

But we can also use a “role option” so that the plugin administrator can choose the role needed for a certain operation.

```
public function ajaxACTION() {
    if (!$this->canUserDoRoleOption('CanDoXXXOperation')) {
        die(1);
    }
    // do operation
    die();
}
```

In this example, ‘CanDoXXXOperation’ is defined as a “Role Option”…a concept in this plugin template code. You can define an option (a.k.a setting) that will appear on your plugin’s option page. You can then set the level of user (Administrator, Editor, Author, Contributor, Subscriber, Anyone) that can access the method. See more about role options.

5. Error Handling

Options you have for reporting errors include:

- Use the die(1) call to indicate an error for calls for data like JSON.
- Silently do no operation in the function. Finish with die();
- If you are returning a message that is displayed on the page, echo the message and call die()

Handling Options

Plugin options (a.k.a. settings) are values that your plugin users can configure in their WP environment. The template code creates a convention for how to deal with options, provides some conveniences for creating and using them easily, and a basic administration page for setting option values. This page will discuss the approach and how different parts work together.

1. Option Prefixing

Part of being a good plugin citizen is avoiding having the same name for an option as any other plugin. Otherwise, they collide in the wp_options table. As a best practice, this plugin template code promotes adding a prefix to all of your option names to avoid collisions. So if your “My Awesome Plugin” has an option like “extra_awesome_mode”, that option should be stored in the DB as “MyAwesomePlugin_extra_awesome_mode”.

Seems a bit verbose. To help, the plugin provides convenience functions to automatically handle prefixing so don’t even see it. To use it, you need to use options handling functions alternatives to the WP stock methods.

WP provides option handling functions: get_option, add_option, update_option, delete_option functions. Instead of using those, you will use functions (methods) defined on your XXX_Plugin.php class: $this->getOption, $this->addOption, $this->updateOption, $this->deleteOption.

will automatically add a prefix to the option name stored in the DB to avoid collisions. $this->getOption also allows you to return a default value if the option is not set.

2. Defining Options

Options are defined declaratively in the XXX_Plugin.php class, getOptionMetaData() function. The purpose of this function is to return an associative array of information about each option. Other parts of the template code use this to do things like automatically generate an administration page.

When you define an option, you can optionally define a selection of values. On the generated options admin page, this will then appear as a drop-down list of choices. Otherwise it appears as a text input field where the user types in a value.

Consider the following example:

```
public function getOptionMetaData() {
    return array(
        //'_version' => array('Installed Version'), // Leave this one commented-out. Uncomment to test upgrades.
        'ATextOption' => array(__('A text option', 'my-awesome-plugin')),
        'Donated' => array(__('I have donated to this plugin', 'my-awesome-plugin'), 'false', 'true'),
        'CanSeeSubmitData' => array(__('Can See Submission data', 'my-awesome-plugin'),
                                    'Administrator', 'Editor', 'Author', 'Contributor', 'Subscriber', 'Anyone')
    );
}
```

Note that all options you define will be automatically prefixed as described above.

The commented-out "_version" line is there for testing. The plugin code tracks which version is installed using this option. If you uncomment that line, you will be able to manually change the version on the options page. You can use this to test upgrade actions because it allows you to set the version back to an older value then the upgrade action will run again. See **Upgrade Actions**.
As you can see, the key of the associative array is the name of option. A prefixed form of this options is stored in the database. The value of an option can be retrieved using $this->getOption($optionName) where $optionName is the same key. All the prefixing stuff is transparent.

The key of the array is associated with an array. The first value in the array is a display string that will be shown on the plugin's options admin page to describe the option. Use the "__()" function with your text domain ('my-awesome-plugin' in the above example) to allow for internationalization of the string (i.e. translation to different languages).

If that associated array has only one value, then the plugin's options admin page will display a text input field where the user can type in a value. Example:

```
'ATextOption' => array(__('A text option', 'my-awesome-plugin')),
```

But if you want a drop-down list of options instead, add them to the array in the order you wish to see them like:

```
 'Donated' => array(__('I have donated to this plugin', 'my-awesome-plugin'), 'false', 'true'),
```

Initializing Option Defaults

But when you do this, there is an issue I call the "**Defaults Issue**". Although you define these options, none of them actually exists in the DB until someone goes to the plugin's options admin page and clicks the save button. At that point all options are saved. Image your code calls

```
if ('false' == $this->getOption('Donated'))
```

before any options are saved. You are expecting the string 'false' but you get null and the "if" test fails.

You can choose to initialize all option defaults during install. The empty initOptions() function is called when the plugin is installed. You could choose to implement this function in XXX_Plugin.php similar to this:

```
protected function initOptions() {
	$options = $this->getOptionMetaData();
	if (!empty($options)) {
		foreach ($options as $key => $arr) {
			if (is_array($arr) && count($arr > 1)) {
				$this->addOption($key, $arr[1]);
			}
		}
	}
}
```

That works just fine if you have all the options defined at install time. But inevitably you will add more options in later versions. Those users who upgrade will not get the default values for new options since the re-install is not re-run. You can set them as part of an **upgrade action**.
Alternatively, you can skip all this initialization. But then you need to handle cases where the options are not already in the Database. You have a couple options here. You can add the default option as an additional parameter to the "getOption" call. So it then becomes:

```
if ('false' == $this->getOption('Donated', 'false'))
```

In this case, if there is no value for 'Donated' in the DB, 'false' is returned (a convenient improvement to the old get_option call). If you do that, then you should ensure that 'false' is the first choice listed in that appreciated array. That way, when a person goes to the plugin's options admin page, 'false' will appear first in the drop-down list but since it is first, it looks like it is selected.

Alternately, you might add an empty string option as the first in you list of choices.

Role Options

The template code provides some conveniences around determining whether or not a user has an adequate role to perform an operation. From the example above:

```
'CanSeeSpecialData' =>
    array(__('Can See Submission data', 'my-awesome-plugin'),
              'Administrator', 'Editor', 'Author',
              'Contributor', 'Subscriber', 'Anyone')
```

These choices correspond to WP roles with the addition of “Anyone” which means a user with no-role (a user who is not logged-in or registered with the WP site). This is a convenient way to provide an option where the plugin administrator can decide what level of user can do an operation.

To enforce this option, you would add code like:

```
if ($this->canUserDoRoleOption('CanSeeSpecialData')) {
    // do protected operation
}
```

In your XXX_OptionsManager.php file, look at the following function to learn more and refer to the **Enforcing Role Security** page.

```
canUserDoRoleOption($optionName)
isUserRoleEqualOrBetterThan($roleName)
getRoleOption($optionName)
roleToCapability($roleName)
```

i18n of Option Choices

In the above example, we use the “__()” function to internationalize the option description but not the values (like ‘true’ and ‘false’). Do NOT use “__()” around the list of values. You don’t want some installations saving ‘true’ but others saving ‘vrai’, ‘evet’, ‘ja’, etc. because when you go to call $this->getOption you don’t know what will be returned.

So we have to find an alternative path to internationalizing those strings. This is done via the $this->getOptionValueI18nString() function defined in the XXX_OptionManager superclass to XXX_Plugin. Have a look at the implementation of the function:

```
protected function getOptionValueI18nString($optionValue) {
	switch ($optionValue) {
		case 'true':
			return __('true', 'TEXT_DOMAIN');
		case 'false':
			return __('false', 'TEXT_DOMAIN');

		case 'Administrator':
			return __('Administrator', 'TEXT_DOMAIN');
		case 'Editor':
			return __('Editor', 'TEXT_DOMAIN');
		case 'Author':
			return __('Author', 'TEXT_DOMAIN');
		case 'Contributor':
			return __('Contributor', 'TEXT_DOMAIN');
		case 'Subscriber':
			return __('Subscriber', 'TEXT_DOMAIN');
		case 'Anyone':
			return __('Anyone', 'TEXT_DOMAIN');
	}
	return $optionValue;
}
```

You will want to add another case to the switch for each value you add as an option choice. You simply add them to that method. A better practice from the OOP perspective is to override that method and implement your own in Plugin_XXX.php. You can call parent::getOptionValueI18nString() to pick up the existing ones.

Options Administration Page

See also: **Adding Administration Pages** wherein it discusses how to turn the options page that you get automatically from this template software into a set of tabs (one of which is the options) where you can add more content.

As described in the **Handling Options** page, options defined in getOptionsMetaData() automatically appear in an administration options page for the plugin.

Let's briefly walk through that mechanism and mention some points were you can make changes.

You will find in your template code, in your XXX_Plugin.php file, this code:

```
public function addActionsAndFilters() {

    // Add options administration page
    add_action('admin_menu', array(&$this,
'addSettingsSubMenuPage'));
```

That add_action call installs the options page under the Plugins menu. More specifically, it calls the addSettingsSubMenuPage() function in the XXX_LifeCycle class.

```
public function addSettingsSubMenuPage() {
    $this->addSettingsSubMenuPageToPluginsMenu();
    //$this->addSettingsSubMenuPageToSettingsMenu();
}
```

Some plugin writers put an options page under the Plugins menu, others put it under the Settings menu. This function chooses the Plugins menu, but you can switch it simply by changing which line is commented out.

The function it calls adds a sub menu page whose content is provided by the settingsPage()

```
protected function addSettingsSubMenuPageToPluginsMenu() {
    $this->requireExtraPluginFiles();
    $displayName = $this->getPluginDisplayName();
    add_submenu_page('plugins.php',
                     $displayName,
                     $displayName,
                     'manage_options',
                     $this->getSettingsSlug(),
                     array(&$this, 'settingsPage'));
}
```

The settingsPage() is defined in XXX_OptionsManager.php. It is a long method. It includes the code for creating a form with fields for all options. If you want to create a different page, override settingsPage() in XXX_Plugin.php.

Enforcing Role Security

Roles
WordPress has notions of roles that can be assigned to users: “Administrator”, “Editor”, “Author”, “Contributor”, “Subscriber”. In your plugin code, you may wish to perform programmatic security, that is to only allow a section of code to be executed if the user is of adequate role.

Your XXX_Plugin class inherits some functions from the XXX_OptionsManager class that make role checking easier. For example:

```
if ($this->isUserRoleEqualOrBetterThan('Author')) {
   // do protected operation
}
```

will indicate if the current user has Author or higher role. This is useful, for example, to put in an “if” statement to guard code that can be executed via an AJAX call. (See more on the **Creating AJAX Calls** page).

Role Options
In some cases, you want the role (‘Author’ in the example above) to be configurable. You can do this by defining a plugin “Role Option” (read how to define one on the **Handling Options** page).
The example above can be changed to:

```
if ($this->canUserDoRoleOption('CanDoSomeSpecialOperation')) {
    // do protected operation
}
```

where ‘CanDoSomeSpecialOperation’ is the name of a role option that you define.

You can get the name of the minimal role level required for a role option using:

```
$this->getRoleOption('CanDoSomeSpecialOperation');
```

Capabilities
In addition to roles, WP has the notion of “capabilities” such as “manage_options”, “publish_pages”, “publish_posts”, “read”. Sometimes you want to call a WP function and it requires a capability parameter of the user. The template code provides a convenience function to convert role to capability.

An example when you want to add an administrative submenu page:

```
$roleAllowed = 'Author';
$capability = $this->roleToCapability($roleAllowed);
```

```
3 add_submenu_page( $parent_slug, $page_title, $menu_title, $capability, $m
enu_slug, $function );
```

Adding Administration Pages

By using this template software, you automatically get an Options page under the Plugins menu. How this is set up is documented in **Options Administration Page**.
But often plugins need more than that rather simplistic options page. One option to to create another administration page. This will create another menu item somewhere in the administration menus. Sometimes that is what you want, but more often what is better is to have one menu item in the administration area for all your configuration information. You don't want users to have to look around for different menu items for different things. It is also not a good idea to unnecessarily pollute the administration menus with lots of stuff.

A second option is to split existing options page into a page with tabs on it, wherein one tab is the stock options page you already get for free, and other tabs have whatever you want on them. I recommend this option for most cases.

Option 1: Creating New Administration Pages & Menu Items

Creating new pages in the administration section is a matter of calling the right WP function to register a function that outputs page contend. These WP APIs are described in **http://codex.wordpress.org/Administration_Menus.**
To use in this plugin, you need to know two things. **First**, you can create a function in your XXX_Plugin class to output such page content. When a WP function parameter takes an input function for a callback, use the object-oriented form of that. In other words, pass "array(&$this, 'yourFunctionName')".
Second, you need to know where to make those calls. The calls to these APIs must originate in the XXX_Plugin::addActionsAndFilters() function. Looking at the template you, you will see the following call:

```
1 public function addActionsAndFilters() {
2     // Add options administration page
3     add_action('admin_menu', array(&$this,
'addSettingsSubMenuPage'));
```

This hooks into the admin_menu action a call to $this->addSettingsSubMenuPage(). In that function a call is made to add_submenu_page to set up a menu item that will display the contents of $this->settingsPage():

```php
protected function addSettingsSubMenuPageToPluginsMenu() {
    $this->requireExtraPluginFiles();
    $displayName = $this->getPluginDisplayName();
    add_submenu_page('plugins.php',
        $displayName,
        $displayName,
        'manage_options',
        $this->getSettingsSlug(),
        array(&$this, 'settingsPage'));
    }
```

Note: if you are creating other pages, you will need to create a “slug” for each that is a unique string to identify your page. In the above, we use the function “getSettingsSlug()”

```php
protected function getSettingsSlug() {
    return get_class($this) . 'Settings';
}
```

Here we use “get_class($this)” as a prefix to another string, ‘Settings’ in this case. The point of this is to add a prefix so that your slug is unique and doesn’t collide with some other plugin’s slugs. I recommend as a best practice to use this “get_class($this) . ‘SomeString’” convention for admin page slugs.

Note also, that in the above add_submenu_page example, we pass in the ‘manage_options’ WP capability name. With this template code, you have the option to make this configurable by defining a “role option” where the plugin administrator can set what level of user (Admin, Editor, etc) gets the menu item. In that case you define a role option (see Handling Options and Enforcing Role Security) the replace **‘manage_options’** above with **$this->roleToCapability($this->getRoleOption(‘TheOptionName’)**

The best way to go about adding administration pages is to look at that example in the code and in Options Administration Page.

Option 2: Changing the Existing Options Page into a set of Tabs

The idea here is that your existing options page is changed into a page with JQuery UI Tabs. One tab is the current options page, the rest of the tabs include content you create. Here are the steps to create this.

First, you will need to get a copy of the JQuery UI CSS. To get it, you need to download JQuery UI and pull it out. Here is a version you can download: jquery-ui.css quickly. Place this file in your plugin's **css/** sub-directory.

In your XXX_Plugin.php file, find the addActionsAndFilters() function. Add the following code. Notice that we are checking the for the options page "slug". This is a way of only adding these styles and scripts if the options/settings page is being viewed. We don't want to add these on all pages.

```
public function addActionsAndFilters() {
        add_action('admin_enqueue_scripts', array(&$this,
'enqueueAdminPageStylesAndScripts'));
}

public function enqueueAdminPageStylesAndScripts() {
    // Needed for the Settings Page
    if (strpos($_SERVER['REQUEST_URI'], $this->getSettingsSlug())
!== false) {
        wp_enqueue_style('jquery-ui', plugins_url('/css/jquery-
ui.css', __FILE__));
        wp_enqueue_script('jquery-ui-core');
        wp_enqueue_script('jquery-ui-tabs');
        // enqueue any othere scripts/styles you need to use
    }
}
```

Second, we override the **settingsPage()** function to output DIVs that will become Tabs and Javascript to initialize the tabs. Use the following template code

(Reference: http://jqueryui.com/demos/tabs/)

```
public function settingsPage() {
       if (!current_user_can('manage_options')) {
              wp_die(__('You do not have sufficient permissions to access
this page.', 'TEXT-DOMAIN'));
       }
       ?>
<div>
       Header section above all the tabs
</div>

```

```
<script type="text/javascript">
	jQuery(function() {
		jQuery("#plugin_config_tabs").tabs();
	});
</script>

<div class="plugin_config">
	<div id="plugin_config_tabs">
		<ul>
			<li><a href="#plugin_config-1">Tab 1 Label</a></li>
			<li><a href="#plugin_config-2">Tab 2 Label</a></li>
			<li><a href="#plugin_config-3">Options</a></li>
		</ul>
		<div id="plugin_config-1">
			<?php $this->outputTab1Contents(); ?>
		</div>
		<div id="plugin_config-2">
			<?php $this->outputTab2Contents(); ?>
		</div>
		<div id="plugin_config-3">
			<?php parent::settingsPage(); ?>
		</div>
	</div>
</div>
<?php
}
```

Notice the call to the superclass parent::settingsPage() to place the stock options page in the content DIV of the third tab.

You would have to create functions like “outputTab1Contents()” and outputTab2Contents() as functions in your XXX_Plugin class. You could simply put the code in-line in their DIVs, but I find this makes the settingsPage() function really long and hard to read. Moving some of the content to other functions makes this more readable. You might also want to put content in another file then use PHP include() to add it.

In the tabs that you create, the best practice is to use AJAX calls to update configuration items from the page instead of doing a form post that does a page transition. See Creating AJAX Calls.

Internationalization

Introduction

Internationalization, or “i18n” for short, is a way of allowing display strings to be changed depending on the language setting of the system. In WP, in the wp-config.php file, you will see something like:

```
define ('WPLANG', '');
```

Which indicates WP is running in the default language (English). If you change this setting to the following, then you would have WP set for Italian.

```
#define ('WPLANG', 'it_IT');
```

In this case, “it” is the Italian language, for “IT” which means Italy. That seems redundant, but it allows you to specify the same language but different varients associated with different countries. Portuguese from Portugal (pt_PT) is bit different from Portuguese from Brazil (pt_BR) for example.

In your WP plugin, you indicate which string are to be “internationalized” (see below). Then you just write your strings in English by convention. You don’t do any translation. What you do is automatically generate a create a file (.pot) that other users can use as a reference to create a translation files (.po & .mo) for a specific language/country (see below). All these files go in your plugin’s languages/ sub-directory.

Once the translation files are in the languages directory of a plugin where it is installed, and that matches the WPLANG setting, then all i18n strings in your plugin will automatically be replaced with what is in the translation file.

If the translator gives those files to you (the plugin writer), you can then include those files in your plugin so that everyone’s installation will have that translation available.

Read more at **Translating WordPress**.

How to I18n Strings in WP

To make a string be i18n, you simply wrap it in the WP double-underscore function “__” and you need a Text Domain configured.

```
$i18nTrue = __('true', 'my-text-domain');
```

If you plan to echo the string, you can use the “_e” function:

```
// This echos a translated string
echo  __('true', 'my-text-domain');
// This does the same thing
_e('true', 'my-text-domain');
```

You should also be aware of how to handle i18n in display string for plugin options. See **Handling Options**.

Setting Up a Text Domain

The template coded sets up a text domain for you automatically. You just need to know what it is.

By default in the template code, it is the same as the name of your plugin directory. Something like “my-awesome-plugin”. It is officially defined in your main plugin file, also named the same as the directory + .php (like “my-awesome-plugin.php”). At the top of that file is a comment section with metadata that WP reads. The Text Domain line is there

```
Text Domain: my-awesome-plugin
```

This should already be set up for you. In addition there is some code in the your XXX_main.php file that initialization your text domain. Have a look if you are curious.

The point of the Text Domain is to have a unique label so that when WP goes to i18n a string, it knows what translations files to use. The translation files in your plugin’s languages/ sub-directory get associated with your Text Domain label.

Creating a POT File

Once you have diligently use the “__” and “_e” function to i18n all your display strings, then all you have to do is generate a POT file and include it in your languages sub-directory.

Assuming you have your plugin registered on WordPress.org (see **Registering Your Plugin on WordPress.org**) all you need to do is go to the Administration tab of your plugin’s page on WordPress.org and it gives you a button to generate a POT file from the code you have checked into SVN. The POT file name will be the same as your text domain + .pot (like “my-awesome-plugin.pot”). Generating a POT file is something to do just before a release. There are more details about generating a POT at **Version and Releases**.

Creating and Sharing Translation Files

In case you want to create translation files yourself, you use the Poedit application to open the POT file. You enter in a translation for each string and save a .po (text) file and a .mo (binary) file. You need to append the language and country abbreviations to the file (same as what would appear in WPLANG).

For example, if you have a “my-awesome-plugin.pot” file and you wanted to translate to Brazilian Portuguese, you would name your files:

- my-awesome-plugin-**pt_BR**.po
- my-awesome-plugin-**pt_BR**.mo

All three files go into the languages/ sub-directory.

For one plugin, I provided a **web page for translators to learn how to use Poedit and how to name files**.

Sneaking Translation Files into Your Version

If a translator shares his translation files with you, you will want to include them in the released version of your software. You can create a new release (new version) with the files included.

See Version and Releases.

But sometimes this seems like something you just want to slip into the current version. You can add the file to the SVN trunk code and to the copy of the code under the current release tag. Anyone who installs/updates to that version after you slipped in the translation files will get them. Those who did it before will not. You can decide for yourself if you think this is an OK practice.

Registering Your Plugin on WordPress.org

The easiest way to share your plugin is to put it on the WordPress.org site. Read the WP Plugin Hosting About Page. Be sure you understand the implications. Highlights are:

- Your code must GPLv2 compatible. This means it is open source.
- You will need to check in your code to the WordPress.org Subversion repository
- You can't do nefarious things

Then when you are ready, apply to add your plugin.

NOTE: In the **Plugin Name** field, you want this name just right. They will take your name like "**My Awesome Plugin**" and create the identifier "**my-awesome-plugin**". This identifier becomes the top level directory of your plugin, and the URL to your plugin on the site will be "http://wordpress.org/extend/plugins/**my-awesome-plugin**/".

When you generate the code for this plugin template, it assumes this will happen. So when you put "My Awesome Plugin" in the form to generate the template code, the generated code is in directory "my-awesome-plugin", and in the my-awesome-plugin.php file in the header at the top, you will have:

```
/*
   Plugin Name: My Awesome Plugin
   Plugin URI: http://wordpress.org/extend/plugins/my-awesome-
plugin/
   Version: 0.1
   Author:
   Description:
   Text Domain: my-awesome-plugin
   License: GPL3
  */
```

So you want to make sure these all match.

Subversion Code Repository

Next you need to get your code into the WP.org Subversion (SVN) repository. Your SVN URL will be something like:

- http://plugins.svn.wordpress.org/my-awesome-plugin/trunk

Check out this directory from SVN, naming it at your plugin's top level directory (like "my-awesome-plugin") instead of "trunk". Put the contents of your existing code directory (like "my-awesome-plugin") into this directory. Add all the files and sub-directories and commit. Make this your working version and commit changes as you make them.
You may want to use a program that helps you use SVN. On Windows, TortoiseSVN is good (and free).

Making a Release

Refer to Versions and Releases. The "Stable tag" line in your readme.txt file (the version of this file on the trunk) tells WP.org which SVN tag has the "released" code. That is what will be available to people in the WP admin consoles.

Support Forum

If you navigate to your Plugin's WP.org home page (like http://wordpress.org/extend/plugins/my-awesome-plugin/) you will find a link to **Forum Posts** (a link like http://wordpress.org/tags/my-awesome-plugin). This is the place where users can post questions and bugs.

Versions and Releases

This is a check-list of things to do to make a release of your plugin through WordPress.org. First you must have your plugin registered on WordPress.org.
We assume here, that your code is checked into the WordPress.org's Subversion (SVN) repository and you are working on the trunk.

1. Give your version a number.

Edit your main plugin file's informational comment header, (file named like "my-awesome-plugin.php"). The header looks like:

```
/*

   Plugin Name: My Awesome Pluin

   Plugin URI: http://wordpress.org/extend/plugins/my-awesome-plugin/

   Version: 0.1

   Author: Author name
```

```
   Description: A description here

   Text Domain: my-awesome-plugin

   License: GPL3

  */
```

Change the “Version: 0.1” to your new Version. Version numbers should be consistent with what the PHP version_compare function recognizes.
Commit this to SVN.

You can do this well in advance of a release. In fact, you probably should update this after you make a release with the next release number that is in development.

2. Update your readme.txt Changelog
Update your readme.txt file with information about your new version in the “Changelog” section. Assuming your new version is “1.4” you would put a

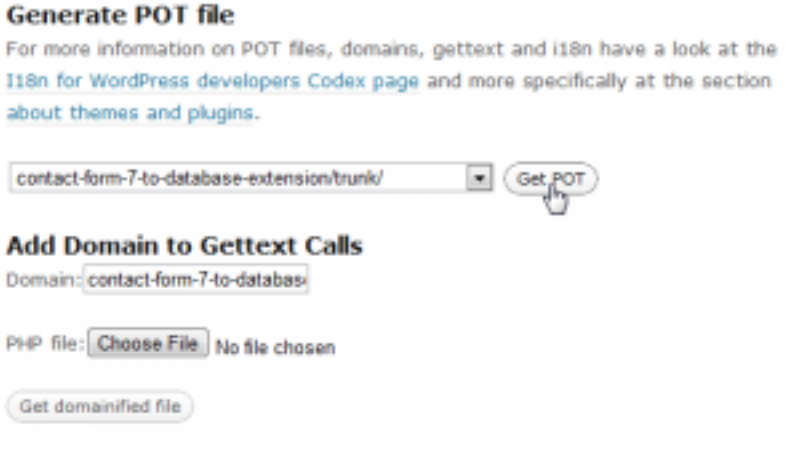
```
== Changelog ==



= 1.4 =

* Cool new feature X added

* Cool new feature Y added
```

Refer to the readme.txt example, markdown format.
Do **NOT** update Stable tag….yet.
Paste the contents in the readme.txt validator to make sure it’s OK.
Commit to SVN

Generally, you should update this file each time you make a noteworthy change to code as you develop it. Don’t wait until you are about to release.

3. Generate an updated .pot file.

This is the reference file that others can use to do translations. Simply go to your plugin's page on WordPress.org, and go to the Admin panel. The URL is something like "http://wordpress.org/extend/plugins/my-awesome-plugin/admin/". The click the **Generate POT** button to create a .pot file from the trunk code.

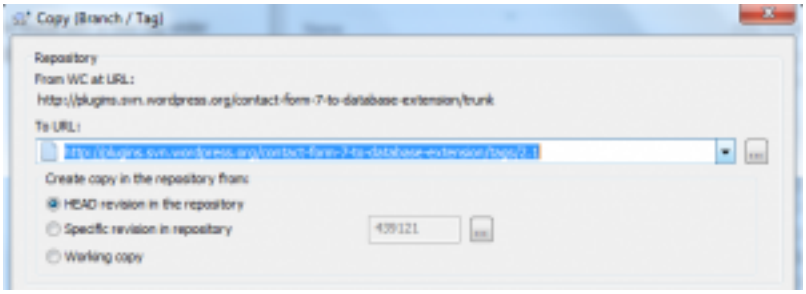


Save the file under your plugin's **languages/** directory. Commit to SVN.

4. Create an SVN Tag

You want to be sure to tag the latest code on the **HEAD of the trunk** with a new tag under "tags". The tag should be a number that is exactly the same as the "Version" number you set in your main plugin file header.

Here is an example Tortoise SVN dialog:



4. Change your Stable Tag in readme.txt

This action actually releases the version to users. The "Stable tag" value signals WordPress.org which version of your software is the released version. Any user installing or upgrading your plugin on their WordPress site will get this version. Once you change the "Stable tag" value those who have your plugin installed will automatically see upgrade notices on their Plugins administrative page. The value of your "Stable tag" should be the SVN tag you created. For example, if you made SVN tag "1.4", you release that version by setting "Stable tag: 1.4" and committing on the ***trunk***.

Point of confusion: Your "Stable tag" is defined in then ***trunk version*** of your **readme.txt** file, ***NOT the tagged version***. So even though your new release is version 1.4 (for example) and you have a "1.4" SVN tag, you will actually be updating the readme.txt on the trunk. The value of "Stable tag" on the 1.4 tagged version is not used.

Edit the ***trunk version*** of your readme.txt file. Change the Stable tag value to match your SVN tag.

```
Stable tag: 1.4
```

SVN Commit the file.

The change will not be immediate on the WordPress.org site. But it should be reflected within an hour (based on my experience). And it will be longer until update notices appear in the Plugins admin page of your user's WP installations.