Golden Dragon

GD Plugin Core

WordPress Plug-in Base Class User Guide

User Guide Version: **0.9.0 Beta**
Widget Version: **1.0.0**
Release Date: **30.july 2008.**

Written by: Milan Petrovic

Official Blog: http://wp.gdragon.info/

Plugin Page: http://wp.gdragon.info/plugins/gd-plugin-core/

Support Forum: http://forum.gdragon.info/viewforum.php?f=17

Support Email: milan@gdragon.info

Contents

Introduction

This is base class that can be used to extend into a plugin.

Basic idea was to shorten the plugin development time. When you start working on a new plugin, you need to create a lot of basic code for initialization of options page, widget initialization and this is even more complicated if you want a plugin to support multi instance widgets. And this you need to repeat for every plugin.

And, repeating this every time gets very tiresome and mistakes are easily made. So, you might want to try using this base class, extending it and setting only the things you need.

Requirements

To use this plugin you must know how the WordPress works, and this means that you are developer, or you want to became one, and you are ready to learn. This is not some magical class that will do all by itself. Class is designed to make creating plugins easier and more streamlined. This user guide, wizard for the plugin and examples with this plugin are here to help you understand how to use it.

After few tries, and tweaking you will be able to create plugins with base class use, and you will see how easy that can be and how much work you can actually skip by doing so.

WordPress

Plugin is developed for newer versions of WordPress and is tested only with 2.5 and 2.6 branches. Most of the features should work with version 2.3 or even older.

User Guide

This user guide is still incomplete.

Class Features

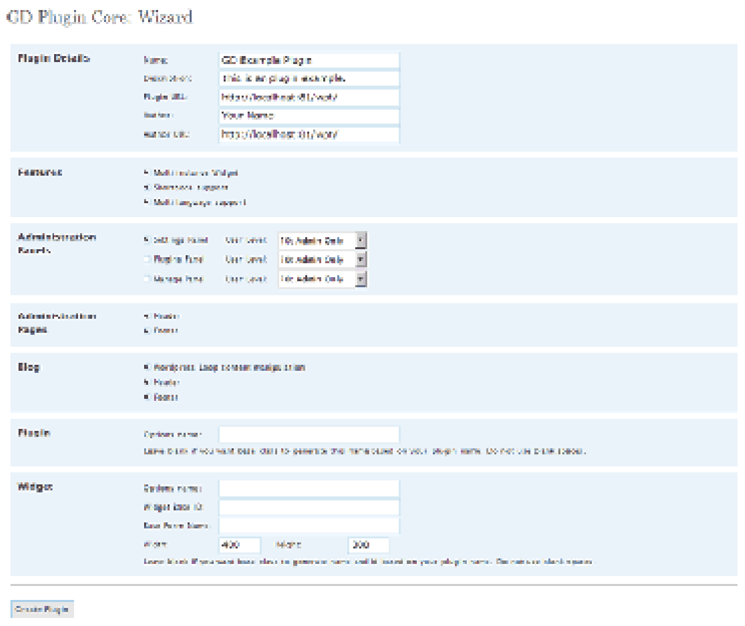
The **GD Plugin Core** class implements following features:

1. Options subpage on a Settings page
2. Options subpage on a Plugins page
3. Options subpage on a Manage page
4. Setting user level for added subpages
5. Multi instance widget
6. Loading of translations if available
7. Adding files (css, js...) into header
8. Event log with log file support
9. WordPress filters for header and footer
10. Functions for generating valid form elements
11. Unlimited number of Shortcodes
12. Additional Helper class with useful functions

For the future development, I am planning to add some more features:

1. Top level options admin page with own subpages
2. Adding more filters and actions
3. Adding more than one subpage for Settings, Plugins and Manage pages
4. Adding subpages in Design page
5. Getting HTML for more form elements
6. More examples

Class Creation Wizard



GD Plugin Core: Wizard

Plugin Details	Name:	GD Example Plugin
	Description:	This is an plugin example.
	Plugin URL:	http://localhost:81/wp/
	Author:	Your Name
	Author URL:	http://localhost:81/wp/
Features	Multi instance Widget	
	Shortcode support	
	Multi language support	
Administration Panels	Settings Panel — User Level:	10, Admin Only
	Plugins Panel — User Level:	10, Admin Only
	Manage Panel — User Level:	10, Admin Only
Administration Pages	Header	
	Footer	
Blog	Wordpress Loop content manipulation	
	Header	
	Footer	
Plugin	Options name:	
	[illegible]	
Widget	Options name:	
	Widget Base ID:	
	Base Form Name:	
	Width: 400	Height: 300
	[illegible]	

Create Plugin

Also, along with this base class plugin, there is one more plugin available in the package. This plugin can be used to create another plugin that will use base class, much like the wizard. You can set a lot of parameters, and the wizard will create new file with your plugin class. This class will have basic function implementation based on your settings.

After that, you can use the file created and add your own code. So, the plugin development time will be shortened considerably. You can concentrate only on the actual coding, and avoid all the basic work plugin requires.

When you activate plugin, you will see **GD Plugin Core Wizard** subpage on **Plugins** page. Since this Wizard is going to create all needed files, you must allow write rights in folder **'plugins-wizard'** located in folder with plugin. Until you do that, **Create Plugin** button on the Wizard page will not be visible, and you will see message that reminds you what you need to do. To use base class you don't need to activate plugin, activate it only if you need to use Wizard.

Only required field for a wizard is Plugin Name. All settings are already set to default values. Depending on the options you have selected, wizard will create up to 5 files and 1 folder:

- **Main Plugin file**: this file contains your plugin class. In every overridden method you have a comment that instructs you where you should put your code in. All the code added is required, and its best not to change what wizard creates. Just add your code in created file and you are set.
- **Widget Form file**: this file should contain form that represents controls for the widget. Here you can see what the form input and select tag need to have for the widget to work. More on this later.
- **Panel Form files**: these are the forms used by the options pages. This page requires full form code unlike the widget. That's way the wizard will create form tag and submit button. More on this later. All 3 panel pages have separate form files.

As you can see, the wizard is very simple to use, and after a few clicks you have a skeleton or blueprint if you like of your plugin.

For the future Wizard versions I am planning some more features:

1. Adding settings for widget and plugin
2. Adding list of files to be loaded in header
3. Adding shortcodes
4. Adding TinyMCE3 buttons
5. Storing data in database for reuse

Best way to see how easy is to use this base class is to actually use the wizard, create few example plugin blueprints and try to add simple features to it.

Class Elements

Variables

Class implements various variables. Because of the class model in PHP, every variable or function from base class can be overridden in the extended class. But only some of the variables in this class should be overridden.

Don't override

Here is the list of variables you should not change in your extended class:

- **$version**: GD Plugin Core class version
- **$events**: array that will be used to store events log
- **$wp_version**: constructor will store WordPress version.
- **$plugin_arguments**: array of arguments from constructors arguments string. More on this will follow.
- **$plugin_name**: name of the plugin, from constructor call
- **$plugin_id**: name of your plugins folder
- **$plugin_url**: full URL to a plugin folder
- **$plugin_path**: full server path to a plugin folder
- **$file_path**: path to your extended class file, passed in constructor call
- **$global_widget_options**: widget options for all widget instances.
- **$options_plugin**: plugin settings from the WordPress options table.
- **$options_widget**: widget settings to current widget instance.
- **$options_widget_number**: id number of a current widget instance.

I will explain all of these variables later in more details.

Override (if you want to, need to or have to)

And here are the few variables that you actually can and usually need to override.

- **$log_file**: path to a file you want to use for logging.
- **$default_options_plugin**: this is array with default settings for your plugin
- **$default_options_widget**: this is array with default settings for the widget
- **$extra_files_admin**: array with files to be added into header of WordPress admin pages
- **$extra_files_db**: array with files to be added into header of WordPress blog pages
- **$shortcodes**: array with definition of shortcodes you want to use and implement

Constructor

Constructor will initialize all needed parameters based on the variables you pass in constructor call. Based on WordPress version plugin base URL will be set. From WP 2.6 getting URL will be different than the previous WP versions. All action and filter will be also added in constructor.

Functions

Initialization
This is the WordPress action implementation for enabling various features. You need to override some of these functions.

Helpers
These functions are used by other functions. In most cases you would not need to use or change these functions.

Events & Log
Here you can find function that will add events into log and file log. It's best not to override any of these functions. Dump functions can be used to dump whole objects into the log file, or all events recorded by the class.

Widget
Here you will find 3 functions needed for a widget to work. Since the widgets implemented by this base class are multi instance widget, it's best not to mess with these functions. Please, read instructions for overriding this function to get widget to work.

Widget Form Elements
Contains functions for generating name and id for form elements, and functions for generating complete form elements. Currently there is only hidden field support added. More to follow soon.

Options Panels
These functions are initially empty, and are intended to be overridden. Here you will implement control panels for your plugin. There are three panels available:

Settings Page
This is submenu options and page in **Settings** page

Plugins Page
This is submenu options and page in **Plugins** page

Manage Page
This is submenu options and page in **Manage** page

Configuration Parameters
These parameters are passed to constructor in form of a string. In string all values are connected with & symbol.

List of parameters

Here are parameters you might want to use for setting the plugin. Most parameters are Boolean, and can be set to 0 or 1. By default all of them are set to 1. And of course, 1 is for enabling and 0 for disabling the feature.

Features:

- **widget**: plugin will have multi instance widget.
- **i18n**: plugin will use multi language support, and base class will load **MO** file if it is available for the active language.
- **shortcode**: enables support for content shortcodes you can use to implement various new commands into WordPress content rendering.

Admin Panel:

- **admin_head**: plugin will add contents into html header for admin pages
- **admin_footer**: plugin will add contents into html footer for admin pages

Admin Pages:

Parameters in this section have different type of value. If is set to 0, then the page is not activated. If you want to activate page, you set the value for the parameter to anything on scale of 1 to 10. This number will represent the user level that will be able to see the created page. If you want only admin to see it you must use value 10.

For other user levels you can check WordPress Codex documentation. Subscriber can't see any of these pages, and you can't set it otherwise.

- **admin_page_settings**: plugin will have options sub page on the WordPress admin settings page. Default value is 10.
- **admin_page_plugins**: plugin will have options sub page on the WordPress admin plugins page. By default, this is disabled.
- **admin_page_manage**: plugin will have options sub page on the WordPress admin manage page. By default, this is disabled.

WordPress Main:

- **wp_head**: plugin will add contents into html header for WordPress pages
- **wp_footer**: plugin will add contents into html footer for WordPress pages
- **wp_content**: plugin will add filter into WordPress loop for post or page

Widget:

- **widget_options**: name that will be used to store settings for widget in database. Default value is derived from plugin name.
- **widget_width**: width for the widget control panel
- **widget_height**: height for the widget control panel
- **widget_idbase**: internal id used for each instance of the plugin. Default value is derived from plugin name.

- **widget_form**: base name for each form element. Each input or select tag must be named using this string. Naming conventions will be explained in details later.

Plugin:

- **plugin_options**: name that will be used to store settings for plugin in database. Default value is derived from plugin name.

Examples

Here are some examples of the string with parameters.

No multi language support:

```
i18n=0
```

Only widget without multi language support:

```
admin_options=0&admin_page_plugins=0&i18n=0
```

Widget with multi language support, custom width and height and custom form name id:

```
admin_page_settings=0&admin_page_plugins=0&widget_width=300&widget_height=200&widget_form=gdco
re
```

Using Base Class

WordPress required comment

WordPress requires for a plugin having a description comment with plugin name, author, and version and so on. You can copy this comment part from any other plugin, and then change it to fit your plugin. Or, you can use Wizard, and you will get valid comment.

Include

First of all, you need to include file(s) with base class so your plugin can extend it.

```
require_once (dirname(dirname(__FILE__)).'/gd-plugin-core/gd-plugin-core.php');
require_once (dirname(dirname(__FILE__)).'/gd-plugin-core/gd-plugin-xtra.php');
```

If you don't want to use the extra helper class, you can omit second **require_once** call. It's very important to use **require_once**, because there could be other plugins that use this base class, and this way you will avoid class re-declaration.

Constructor

If you look at base class constructor, you will see that it has three parameters, with two first required:

- **plugin name**: name of your plugin, it will be used by base class
- **file path**: this always need to be **__FILE__**
- **parameters**: this is a string with one or more parameters from the list in the previous chapter

This basic class looks like this:

```
class GDCoreExample extends GDPluginCore
{
    function GDCoreExample() {
        parent::GDPluginCore('GD Core Example', __FILE__);
    }
}
```

If you don't use parameters, then plugin will be created with default settings. After the parent call in the constructor, you can add your own code for something's that are not covered by the base class. This could be processing of post or get requests or anything else you might need.

Default Settings

I think that is a good practice to have default settings for a plugin listed in an array in the class. This way you have all your settings in one place, all default values also, and is easier to maintain.

Also, this approach is also important for the widget multi instance support to work. When a new instance of the widget is added, we need to pass default values to the options variable for the widget form to use.

Since base class has implemented full widget support, you need to add default settings in override variable **$default_options_widget**. This variable is an array with named elements like in example below. Base class will automatically update settings stored for widget in database when you add or remove some of the settings from this list, and the stored settings will always be up to date.

```
var $default_options_widget = Array(
    "title" => "Example",
    "message" => "This is an core plugin example widget."
);
```

This is required only for widget part of the plugin. As you have noticed there is one more variable **$default_options_plugin**. This variable has similar use as the widget variable but for the settings that plugin will use but they are not related to a widget part of the plugin. This variable is not used by the base class, but also, it's good to have same system for storing settings as with the widget.

Extra files (CSS and JS)

In most cases you will need to use your own JavaScript and css files in both blog pages and admin pages. For loading these files WordPress uses **wp_head** and **admin_head** actions. Base class implements creation of html tags to link these extra files, and you only need to specify them. This is also done with use of arrays and overriding two class variables, one for admin header (**$extra_files_admin**) and the other for blog (**$extra_files_wp**).

Each of these arrays contains list of files. And each of these files can be stored in two different formats. One is regular string, and it contains relative path to a file you want to include (relative to your main plugin php file). Based on the file name base class will add appropriate html code.

The other format is an array with few more parameters. In this case at least first two elements of array must be used:

- **name**: relative path to a file (must include extension)
- **ext**: extension of the file, used to determine the format to add a file.
- **media**: media code for the link tag
- **type**: type code for the link tag
- **rel**: rel code for the link tag

There are two types of external file html tags. One is used for JavaScript (SCRIPT tag) and the other for CSS and all other external files (LINK). Link file requires some additional attributes, and for that you need to use array file definition. For CSS all elements can be generated automatically. But, for instance if you want to add CSS that doesn't have media attribute to screen (default), you must use array format:

```
array(name => "print.css", ext => "css", media => "print");
```

If you don't set some of the elements, then default value will be used.

Shortcodes

Shortcodes are custom tags you can use to modify the post or page output. Base class can add as many shortcodes as you want. Important thing to know is that every shortcode requires one function in the class that will be used to transform the code into an output. Shortcodes are added into an array.

Once again, each shortcode can be added as a string or as an array. If you add it as a string, then string is your shortcode, and base class will create call back function. Shortcode function will be named **'shortcode_<your-code-name>'**. Shortcode must only contain alphanumeric characters, no spaces.

Array format for a shortcode contains only two strings:

- **name**: shortcode name
- **function**: shortcode function in your class

Important thing is that shortcode function must exist in your plugin class, or you will get an error from WordPress.

Setting the Widget

This is the trickiest part of using the base class. Since multi widget support is rather complicated, you must follow instructions exactly. In your plugin you need to override tall three base class functions for widgets. But you need to do it in the exact way as it's here:

```
function widget_control($widget_args = 1) {
    parent::widget_control($widget_args);

    include "widget_form_file.php";
}

function widget_parse_options($post_params) {
    $options = parent::widget_parse_options($post_params);

    $options['title'] = $post_params['title'];
    $options['message] = $post_params['message'];
    ...
    ...

    return $options;
}

function widget_display($args, $widget_args = 1) {
    parent:: widget_display($args, $widget_args);
    extract($args);
    $options = $this->options_widget;

    // your code here
}
```

Widget Control

First function is the main widget function. To ensure that multi instance support works, you first must call parent function. After that, class variables **$options_widget** and **$options_widget_number** contain values for that specific instance of the widget. For the code to be easier to follow, complete form with controls for a widget is best to be in separate file in the example above called **'widget_form_file.php'**. I recommend using external files for all forms, because it's easier to track changes you make, and your code is much easier to read and follow.

To make it even easier to use, we need to include the class instance using php global. Then you can use build in functions to generate id and names for form elements or even generate complete elements. Example for widget form file in the plugin package contains widget form file, and you can take a look at that. Because name and id for each element must be in a specific format, it's best to use build in functions to get that values (**$pc_GDE** is global instance of your plugin, see included example):

```
name = "<?php $pc_GDE->widget_element_name('option_name'); ?>"
id = "<?php $pc_GDE->widget_element_id('option_name'); ?>"
```

Id can have any value, but I like using this because it's not confusing, and have similar structure to a name of the element. Option name in both lines is actual name of the setting for your widget as it appears in the options array. Also, first element of the form must be a base class required hidden input element. It's best to use build in functions to get these two hidden fields.

```
<?php $pc_GDE->widget_render_hidden('wcore', 'wcore'); ?>
```

And last element is a submit part:

```
<?php $pc_GDE->widget_render_hidden('submit', 'submit'); ?>
```

In between these two is your form. For explanation of the name and id, see the next paragraph.

Widget Parse Options

Second option you have overridden is for parsing the post response from your widget controls. As you have seen above, every form element corresponds to one setting for your widget. Now, each of this form elements values needs to be saved into options array.

Example for this function calls parent function to get **wcore** setting and creates empty array. Now you have set **$options** array and you add values from the post for each setting. Variable passed to this function **$post_params** is normalized and it is array that only contains actual setting names and that is the last part of your element names. As you see above, each element name is made of three parts:

```
formid[number][setting]
```

Formid and Number are generated by base class. Example shows you how to save two settings for title and message.

Widget Display

This is the last part of the widget implementation. This is actual code that's used to show your widget to a blog visitor. Base class only implements one thing in the chain of widget display: and that is getting settings for a specific widget instance. So, you have your options in **$options** array variable and you work with that. Extract arguments call is still needed to get widget parameters generated by WordPress. These parameters include values you need to use to render widget properly (before title, after title…).

Additional Administrator Pages

Base class can be used to create menu options for different administration pages in WordPress.

Instance

After your class definition, you need to create one class instance, so the code will be loaded by the WordPress.

```
$pc_GDE = new $GDCoreExample();
```

Examples

Plugin examples included in the package are very simple, but it implements features described in this user guide. Each new version will have new examples if it is needed to demonstrate new features.

GD Example Plugin I

This is first plugin that will only widget implemented.