Plugin Everywhere. Plugin Writing Tutorial

Author: Yuriy Zisin <devel@yzisin.com>
Date: 2008-10-28
Pages: 7

Document info

This document contains basic information about the plugin subsystem and the tutorial showing how to write your own plugin.

Table of contents

1 Plugin subsystem....1
1.1 Hooks....1
1.2 Plugin Manager....2
1.3 Operating objects....2
1.3.1 Sensors....2
1.4 Static objects transport layer....3
2 Simple plugin....3
2.1 Preparing classes....3
2.2 Enabling plugin....4
2.3 Initial check....4
2.4 Setting hooks....4
2.5 Running live....5
2.6 Using Sensors....6
3 Conclusion....6

1 Plugin subsystem

Plugin subsystem works as a separate level and doesn't require you to change the existing code. It works transparently and supports only two events – event before any object/ method call and event after the call. This is enough to make plugins work flexible and extend basic functionality. Plugin system core mechanism is called “Hooks”.

1.1 Hooks

Hook is a simple and flexible way to integrate plugins into the existing code without its changing. What is hook – hook is a catch mechanism where you can get your handler running by a specified event. It means when existing code is working and reaches a point you set your handler to (you did hook point), main thread runs your handler and allows you to perform some operations that can affect main thread behavior.

So, if you need to apply fixed discount to user's shopping cart, you can set a hook to a method which calculates totals and then apply the discount.

Hooks are under Plugin Manager control.

Author: Yuriy Zisin <devel@yzisin.com>
Date: 2008-10-28
Pages: 7

1.2 Plugin Manager

Plugin manager is a heart of the plugin subsystem. It's tasks are to load and initialize plugins, to provide callbacks and API for plugin classes, etc. Plugin Manager implemented with Singleton pattern and this allows you to reach it anywhere in the code.

As long as Plugin Manager provides transparent plugin support, it also provides you access to any existing project classes, constants, static methods, etc. Honestly, it just doesn't restrict plugin to to do that.

1.3 Operating objects

Each plugin handler method should have two arguments: “object” – the object raised an event, and “parameters” - parameters that were passed to (returned from) the original method. What kind of parameters you will get in handler depends on when you set to call the handler. You will get incoming parameters if you set to call handler before the original method, and outgoing if to call after.

So, you have the parameters original method takes or the return value it produces. At this point you can affect the data flow and apply changes. You also have the original object instance and you can easily use it's getter, setters, public methods, etc.

But in addition, Plugin Manager provides you Sensors API on the original object.

1.3.1 Sensors

Sensors allow you to access ALL object properties. Yes, all of them, even private ones. This mechanism is needed to reduce potential plugin overload. For example, you have to get a lot of data to perform the calculations. And you have to get the same data original object already has, but doesn't allow others to get it. Sensors will allow you to access this information and help you to avoid database queries, cache operations, etc. that original object already did.

Here is handler example:

```
public function onBeforeCall($object, $parameters) {
        $data = $object->getSensorData('_privateData');
}
```

As you can see, this handler does nothing except accessing private object property $_privateData.

Author: Yuriy Zisin <devel@yzisin.com>
Date: 2008-10-28
Pages: 7

1.4 Static objects transport layer

All above works for dynamically created objects. But you may have to write a plugin using static object events. What will happen in this case? Nothing! You will not feel any difference and your handler will work for both static and dynamic objects without any changes.

It is implemented via Plugin Transport layer. This layer performs statical objects wrapping and just delegates your calls to the real static object. Plugin handler method give an instance of “PluginTransport” class that is already tuned to bridge your plugin with the original static object. So, the example above will work in both situation.

Static objects also have Sensors API enabled and work in the same way.

2 Simple plugin

Ok, theory is enough, because it doesn't help much to write your own plugin.

TASK: Write a plugin which will add 10 to the result of the object method. We will experiment on the following class objects:

```
class TestClass {
    private $_privateData = 100;
    public function returnMe($val) {
        return $val;
    }
}
```

As you see, it just returns the value you passed.

2.1 Preparing classes

Each plugin should be extended from “PluginBase” class. In other case it won't even load.

Create plugin stub as shown below:

Author: Yuriy Zisin <devel@yzisin.com>
Date: 2008-10-28
Pages: 7

```
class TestPlugin extends PluginBase {

  public function setStartupHooks()  {
  }

  public function setEventHooks()   {
  }

}
```

and save it as “TestPlugin.php” in the place you want to store plugins.

setStartupHooks – this is the method that allows you to set the hooks to the specified places in project and get an action when project has all required data. For example, you need user identity object to obtain some user data and this should be done before any of event you wish to observe. In this case you should set startup hook to the place exactly after the identity object is created.

setEventHooks – this is the method where you set all hooks to all methods you interested in calls from.

Basically there is no difference between these two methods, just to separate logic.

2.2 Enabling plugin

No plugins are loaded by default. You should take care about loading necessary plugins. You can do it in the following way:

```
 //Plugin subsystem
require 'lib/PluginManager.php';
require 'lib/PluginBase.php';
require 'lib/PluginTransport.php';

//You should require the plugin file for PluginManager to find the plugin class
require 'TestPlugin.php';
PluginManager::getInstance()->loadPlugin('TestPlugin'); //We tell manager class name only
```

Author: Yuriy Zisin <devel@yzisin.com>
Date: 2008-10-28
Pages: 7

2.3 Initial check

Once you done with fresh plugin stub, try to access the application. You shouldn't get any errors, warnings or other unusual behavior. If so – you plugin should work properly.

But it doesn't perform any action now.

2.4 Setting hooks

Now we are ready to set the hooks. Because we don't have any specific tasks for plugin, we can omit setStartupHooks method and leave it empty.

So, what should we do to receive events from our test object? We should watch for “returnMe” method events.

To watch it, you should request Plugin Manager to set the hook. Change your “setEventHooks” and make it looking as below:

```
public function setEventHooks()   {
        PluginManager::getInstance()->setHook(
                $this,
                'onReturnMe',
                PluginManager::HOOK_ON_AFTER_CALL,
                'TestClass',
                'returnMe'
        );
}
```

As you can see – it's simple to set a hook to the specified event. The code above can be explained as:

Set hook for TestClass::returnMe method and call onReturnMe method of $this object after the original returnMe method is called.

Now you should define onReturnMe method because it was specified as a handler. It's time to think what should this handler do. Because we decided just to add 10 to the result original method returns, define the onReturnMe method as:

```
public function onReturnMe($object, $parameters)   {
        $parameters += 10;
        return $parameters;
}
```

If you change something in handler, you also have to return changed parameters to allow your changes to go forward. Be sure you return the same structure and type you've got as an input. PluginManager watches the types you operate with and restricts changes in case you return value with type different from the original method.

2.5 Running live

Ok, it's time to check if everything works. Make a sample script and call test method. Like this:

```
 //Plugin subsystem
require 'lib/PluginManager.php';
require 'lib/PluginBase.php';
require 'lib/PluginTransport.php';

//You should require the plugin file for PluginManager to find the plugin class
require 'TestPlugin.php';
PluginManager::getInstance()->loadPlugin('TestPlugin'); //We tell manager class name only

//We have to allow PluginManager to enable hooks for TestClass. Usually this a part
//of autoloader
if (!PluginManager::getInstance()->wrapClass('TestClass', 'TestClass.php')) {
//But if it has nothing to do with class, we have to load it anyway
   require 'testClass.php';
}

//Let's just call our method
$tst = new TestClass;
echo "\n\n".$tst->returnMe(1)."\n\n";
```

The example above should output “11”. If so – then your plugin works fine.
Comment the line where plugin gets loaded and try again – it will print “1” which is the original method result.

2.6 Using Sensors

To understand the Sensors API, change the handler as follows:

Author: Yuriy Zisin <devel@yzisin.com>
Date: 2008-10-28
Pages: 7

```
public function onReturnMe($object, $parameters)   {
        $parameters += $object->getSensorData('_privateData');
        return $parameters;
 }
```

Once it is done, the return value will be incremented by the value in $_privateData property. And it's all done in so simple way!

3 Conclusion

The plugin subsystem provides a flexible way to easily extend project functionality and it's not so hard to build your own plugin. But it has several side effects. Some of them are already known, but there could be something I don't know yet. What is known:

- __FILE__ constant doesn't work.
- DOMDocument object stops working somehow. It remains DOMDocument object, but doesn't want to output valid XML. It outputs only contents, not tags.
- Not every class could be used to set hooks for. It depends on how this class is built. No concrete rules, you can just try.