Plugins

Overview

Joomla! plugins serve a variety of purposes. As modules enhance the presentation of the final output of the Web site, plugins enhance the data and can also provide additional, installable functionality. Joomla! plugins enable you to execute code in response to certain events, either Joomla! core events or custom events that are triggered from your own code. This is a powerful way of extending the basic Joomla! functionality.

See Joomla Events for a list of core Joomla plugin events.

See Component Events for a list of Hubzero plugin events.

Core Types

Plug-ins are managed at a group level that is defined in the plug-in's XML manifest file. While the number of possible types of plugins is almost limitless, there are a number of core plugin types that are used by Joomla!. These core types are grouped into directories under /plugins. They are:

- authentication
- content
- editors
- editors-xtd
- search
- system
- user
- xmlrpc

Authentication
: plugins allow you to authenticate (to allow you to login) against different sources. By default you will authenticate against the Joomla! user database when you try to login. However, there are other methods available such as by OpenID, by a Google account, LDAP, and many others. Wherever a source has a public API, you can write an authentication plugin to verify the login credentials against this source. For example, you could write a plugin to authenticate against Twitter accounts because they have a public API.

Content
: plugins modify and add features to displayed content. For example, content plugins can cloak email address or can convert URL's into SEF format. Content plugins can also look for markers in content and replace them with other text or HTML. For example, the Load Module plugin will take {*loadmodule banner1*} (you would remove the *'s in practice. They are included to actually prevent the plugin from working in this article),

load all the modules in the banner1 position and replace the marker with that output.

Editor
: plugins allow you to add new content editors (usually WYSIYWG).

Editor-XTD
: (extended) plugins allow you to add additional buttons to the editors. For example, the *Image*, *Pagebreak* and *Read more* buttons below the default editor are actually plugins.

Search
: plugins allow you to search different content from different components. For example, search plugins for Articles, Contacts and Weblinks are already provided in Joomla!.

System
: plugins allow you to perform actions at various points in the execution of the PHP code that runs a Joomla! Web site.

User
: plugins allow you to perform actions at different times with respect to users. Such times include logging in and out and also saving a user. User plugins are typically user to "bridge" between web applications (such as creating a Joomla! to phpBB bridge).

XML-RPC
: plugins allow you to provide additional XML-RPC web services for your site. When your Web site exposes web services, it gives you the ability to interact remotely, possibly from a desktop application. Web services are a fairly advanced topic and will not be covered in much detail here.

Directory & File Structure

While a plugin can have any number of files, there are two you need as a minimum and there are specific naming conventions you must follow. Before we look at the files, we must decide what sort of plugin we are going to create. It must either fall under one of the built-in types (authentication, content, editors, editors-xtd, search, system, user or xmlrpc) or your can create your own type by adding a new folder under /plugins. So, files for an authentication plugin will be saved under /plugins/authentication, files for a system plugin will be saved under /plugins/system, and so on.

The typical plugin install location and files:

```
/hubzero
  /plugins
    /{PluginType}
      {PluginName}.php
      {PluginName}.xml
```

As mentioned, a plugin has a minimum of two files: a PHP file, test.php, which is the file actually loaded by Joomla! and an XML file, text.xml, which contains meta and installation information for the plugin as well as the definition of the plugin parameters.

There is no restriction on the file name for the plugin (although we recommend sticking with alpha-numeric characters and underscores only), but once you decide on the file name, it will set the naming convention for other parts of the plugin.

Examples

A plugin demonstrating basic setup:

Download: System Test plugin (.zip)

Installation

Installing

See Installing Extensions for details.

Uninstalling

See Uninstalling Extensions for details.

Manifests

Overview

All plugins should include a manifest in the form of an XML document named the same as the plugin. So, a plugin named test.php would have an accompanying test.xml manifest.

Directory & Files

Manifests are stored in the same directory as the plugin file itself and must be named the same (file extension being the obvious exception).

```
/hubzero
  /plugins
    /{PluginType}
      {PluginName}.php
      {PluginName}.xml
```

Structure

A typical plugin manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<install version="1.5.2" type="plugin" group="system" method="upgrade"
>
 <name>System - Test</name>
 <author>Author</author>
 <creationDate>Month 2008</creationDate>
 <copyright>Copyright (C) 2008 Holder. All rights reserved.</copyright
>
 <license>GNU General Public License</license>
 <authorEmail>email</authorEmail>
 <authorUrl>url</authorUrl>
 <version>1.0.1</version>
 <description>A test system plugin</description>
 <files>
  <filename plugin="example">example.php</filename>
 </files>
 <params>
    <param name="example"
     type="text"
     default=""
```

```
     label="Example"
     description="An example text parameter" />
 </params>
</install>
```

Let's go through some of the most important tags:

INSTALL
: The install tag has several key attributes. The type must be "plugin" and you must specify the group. The group attribute is required and is the name of the directory you saved your files in (for example, system, content, etc). We use the method="upgrade" attribute to allow us to install the extension without uninstalling. In other words, if you are sharing this plugin with other, they can just install the new version over the top of the old one.

NAME
: We usually start the name with the type of plugin this is. Our example is a system plugin and it has some some nebulous test purpose. So we have named the plugin "System - Test". You can name the plugins in any way, but this is a common format.

FILES
: The files tag includes all of the files that will will be installed with the plugin. Plugins can also support be installed with subdirectories. To specify these just all a FOLDER tag, <folder>test</folder>. It is common practice to have only one subdirectory and name it the same as the plugin PHP file (without the extension of course).

PARAMS
: Any number of parameters can be specified for a plugin. Please note there is no "advanced" group for plugins as there is in modules and components.

See Joomla!'s Documentation on the full list of available parameter types and what they do.

Controllers

Overview

All plugins will have a primary class extending JPlugin that contains the logic and events to be triggered.

Directory & Files

Plugin files are stored in a sub-directory of the /plugins directory. The sub-directory represents what type the plugin belongs to. This allows for plugins of the same name but for different types. For example, one could have a plugin named example for both the /system and /search types.

Note: plugins will always be within a type sub-directory and will never be found in the top-level /plugins directory.

```
/hubzero
  /plugins
    /{PluginType}
      {PluginName}.php
      {PluginName}.xml
```

There is no restriction on the file name for the plugin (although it is recommended to stick with alpha-numeric characters and underscores only), but once you decide on the file name, it will set the naming convention for other parts of the plugin.

Structure

Here we have a typical plugin class:

```
<?php
// no direct access
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport( 'joomla.plugin.plugin' );

/**
 * Example system plugin
 */
class plgSystemTest extends JPlugin
```

```
{
 /**
  * Constructor
  *
  * For php4 compatibility we must not use the __constructor as a cons
tructor for plugins
  * because func_get_args ( void ) returns a copy of all passed argume
nts NOT references.
  * This causes problems with cross-
referencing necessary for the observer design pattern.
  *
  * @access protected
  * @param object $subject The object to observe
  * @param  array   $config  An array that holds the plugin configurat
ion
  * @since 1.0
  */
 function plgSystemTest( &$subject, $config )
 {
  parent::__construct( $subject, $config );

  // Do some extra initialization in this constructor if required
 }

 /**
  * Do something onAfterInitialise
  */
 function onAfterInitialise()
 {
  // Perform some action
 }
}
```

Let's look at this file in detail. Please note that the usual Docblock (the comment block you normally see at the top of most PHP files) has been omitted for clarity.

The file starts with the normal check for defined('_JEXEC') which ensures that the file will fail to execute if access directly via the URL. This is a very important security feature and the line must be placed before any other executable PHP in the file (it's fine to go after all the initial comment though). The importance of having this check your PHP files cannot be over-emphasised.

Next we use the jimport function to load the library file with the definition of the JPlugin class.

You will notice that a plugin is simply a class derived from JPlugin (this differs from previous versions of Joomla!). The naming convention of this class is very important. The formula for this name is:

plg + Proper case name of the plugin directory + Proper case name of the plugin file without the extension.

Proper case simply means that we capitalise the first letter of the name. When we join them altogether it's then referred to as "Camel Case". The case is not that important as PHP classes are not case-sensitive but it's the convention Joomla! uses and generally makes the code a little more readable.

For our test system plugin, the formula gives us a class name of:

plg + **S**ystem + **T**est = plgSystemTest

Let's move on to the methods in the class.

The first method, which is called the constructor, is completely optional. You only require this is you want to do some work when the plugin is actually loaded by Joomla!. This happens with a call to the helper method JPluginHelper::importPlugin(*<plugin_type>*). This means that you even if the plugin is never triggered, for whatever reason, you still have an opportunity to execute code if you need to in the constructor.

In PHP 4 the name of the constructor method is the same as the name of the class. If you were designing only for PHP 5 you could replace this with the name of __constructor instead.

The remaining methods will take on the name of "events" that are trigger throughout the execution of the Joomla! code. In the example, we know there is an event called onAfterInitialise which is the first event called after the Joomla! application sets itself up for work. For more information on when some events are triggered, see the API Execution Order page on the Documentation Wiki.

The naming rule here is simple: the name of the method must be the same as the event on which you want it triggered. The Joomla! Framework will auto-register all the methods in the class for you.

That's the basics of the plugin PHP file. It's location, name and methods will depend on what you want to use the plugin for.

Joomla Events

One thing to note about system plugins is that they are not limited to handling just system events. Because the system plugins are always loaded on each run of the Joomla! PHP, you can include any triggered event in a system plugin.

The events triggered in Joomla! are:

Authentication

- onAuthenticate

Content

- onPrepareContent
- onAfterDisplayTitle
- onBeforeDisplayContent
- onBeforeContentSave (new in 1.5.4)
- onAfterContentSave (new in 1.5.4)

Editors

- onInit
- onGetContent
- onSetContent
- onSave
- onDisplay
- onGetInsertMethod

Editors XTD (Extended)

- onDisplay

Search

- onSearch
- onSearchAreas

System

- onAfterInitialise
- onAfterRoute
- onAfterDispatch
- onAfterRender

User

- onLoginUser
- onLoginFailure
- onLogoutUser
- onLogoutFailure
- onBeforeStoreUser

- onAfterStoreUser
- onBeforeDeleteUser
- onAfterDeleteUser

XML-RPC

- onGetWebServices

For more detailed information on how to create specific plugins, visit the Plugins Category on the Joomla! Documentation Wiki.

Component Events

The following are events that are triggered from within their respective components:

Groups

- onGroupAreas
- onGroup
- onGroupNew
- onGroupDeleteCount
- onGroupDelete

Members

- onMembersAreas
- onMember

Tools

- onBeforeSessionInvoke
- onAfterSessionInvoke
- onBeforeSessionStart
- onAfterSessionStart
- onBeforeSessionStop
- onAfterSessionStop

Resources

- onResourcesAreas
- onResources

Support

- onPreTicketSubmission

- onTicketSubmission
- getReportedItem
- deleteReportedItem

Tags

- onTagAreas
- onTagView

Usage

- onUsageAreas
- onUsageDisplay

What's New

- onWhatsnewAreas
- onWhatsnew

XMessage

- onTakeAction
- onSendMessage
- onMessageMethods
- onMessage

XSearch

- onXSearchAreas
- onXSearch

Languages

Overview

Language translation files are placed inside the appropriate language languages directory within a widget.

```
/hubzero
  /language
    /{LanguageName}
      {LanguageName}.plg_{GroupName}_{PluginName}.ini
```

Note: Plugin language files contain data for both the front-end and administrative back-end.

Setup

As previously mentioned, language files are setup as key/value pairs. A key is used within the plugin's code and the translator retrieves the associated string for the given language. The following code is an extract from a typical plugin language file.

```
; Plugin - System - Test (en-US)
PLG_SYSTEM_TEST_HERE_IS_LINE_ONE = "Here is line one"
PLG_SYSTEM_TEST_HERE_IS_LINE_TWO = "Here is line two"
PLG_SYSTEM_TEST_MYLINE = "My Line"
```

Translation keys can be upper or lowercase or a mix of the two and may contain underscores but no spaces. HUBzero convention is to have keys all uppercase with words separated by underscores, following a pattern of PLG_{PluginGroup}_{PluginName}_{Text} for naming. Adhering to this naming convention is not required but is strongly recommended as it can help avoid potential translation collisions.

See the Languages overview for details.

Loading

The appropriate language file for a plugin is **not** preloaded when the plugin is instantiated as many plugins may not have language files at all. As such, one must specifically load any file(s) if

they are needed. This can be done in the plugin's constructor but is more commonly found outside of the class altogether. Here we see the test plugin for the examples plugins group loading its language file right before declaration of the plugin's class.

```
<?php
// Check to ensure this file is included in Joomla!
defined('_JEXEC') or die( 'Restricted access' );

jimport( 'joomla.plugin.plugin' );
JPlugin::loadLanguage( 'plg_system_test' );

class plgSystemTest extends JPlugin
{
 ....
}
```

Note that the string passed to the loadLanguage() method matches the pattern for the naming of the language file itself, minus the language prefix and file extension.

Translating Text

Below is an example of accessing the translate helper:

```
<p><?php echo JText::_("PLGN_EXAMPLE_MY_LINE"); ?></p>
```

Strings or keys not found in the current translation file will output as is.

See the Languages overview for details.

Layouts

Overview

Note: Plugin views are an additional feature brought through HUBzero libraries. A standard, non-HUBzero Joomla! install will not have this capability.

The majority of plugins will not have view files. Occasionally, however, a plugin will return HTML and it is considered best practices to have a more MVC structure to your plugin and put all HTML and display code into view files. This allows for separation of the logic from presentation. There is a second advantage to this, however, which is that it will allow the presentation to be overridden easily by any Joomla! 1.5 template for optimal integration into any site.

Overriding plugin, module, and component presentation in templates is further explained in the Templates: Overrides section.

Directory Structure & Files

Plugins, like components and modules, are set up in a particular directory structure.

```
/plugins
  /groups
    forum.php   (the main plugin file)
    forum.xml   (the installation XML file)
    /forum
      /views
        /browse
          /tmpl
            default.php   (the layout)
            default.xml   (the layout installation XML file)
```

Similar to components, under the views directory of the plugin's self-titled directory (in the example, forum) there are directories for each view name. Within each view directory is a /tmpl/ directory. There is usually only one layout file but depending on who wrote the plugin, and how it is written, there could be more.

Implementation

Loading a plugin view

```
class plgExamplesTest extends JPlugin
{
    ...

    public function onReturnHtml()
    {
        // Include the HUBzero library that allows plugin views to wor
k
  ximport('Hubzero_Plugin_View');

  // Instantiate a new view
  $view = new Hubzero_Plugin_View(
   array(
    'folder'=>'examples',
    'element'=>'test',
    'name'=>'display'
   )
  );

  // Set any data the view may need
  $view->hello = 'Hello, World';

  // Set any errors
  if ($this->getError()) {
   $view->setError( $this->getError() );
  }

  // Return the view
  return $view->loadTemplate();
    }
}
```

In the example, we're instantiating a new plugin view and passing it an array of variables that tell the object where to load the view HTML from. folder is the plugin group, element is the plugin, and name is the name of the view that is to be loaded. So, in this case, it would correspond to a view found here:

```
/plugins
  /examples
    /test
      /views
        /display
          /tmpl
```

```
            default.php   (the layout)
            default.xml   (the layout installation XML file)
```

Also note that we're returning $view->loadTemplate() rather than calling $view->display(). The loadTemplate() method captures the HTML output of the view rather than printing it out to the screen. This allows us to store the output in a variable and pass it around for later display.

The plugin view file

Our view (default.php) is constructed the same as any module or component view file:

```
<?php defined('_JEXEC') or die('Restricted access'); // no direct acce
ss ?>
<p>
 <?php echo $this->hello; ?>
</p>
```

Packaging

Overview

Packaging a plugin for distribution is easy. If you only have the two files (the PHP file and the XML file), just "zip" them up into a compressed archive file. If your plugin uses a subdirectory, then simply include that in the archive as well.

Loading

Triggering Events

Using the plugin system in your add-on is fairly simple. The most important part is good planning because, to some degree, you're defining an interface for other people to use.

The first thing you need to do is to load your plug-in group. This is done via the following code:

```
JPluginHelper::importPlugin( 'myplugingroup' );
```

This will load all enabled plug-ins that have defined themselves as part of your group. The next thing you need to do is get an instance of the JDispatcher class like so:

```
$dispatcher =& JDispatcher::getInstance();
```

Notice two things here. First, we are using the getInstance() method, not "new" to create a new instance. That is because we need to get the global singleton instance of the JDispatcher object which contains a list of all the plug-ins available. Second, we are using the =& construct to make sure we have a reference to the instance of the JDispatcher and not a copy. Of course this really only applies to PHP version 4, but since you are a good cross-version developer, you will allow for PHP 4 users.

Next, we need to trigger our custom event:

```
$results = $dispatcher->trigger( 'onCdAddedToLibrary', array( &$artist
, &$title ) );
```

Here we have triggered the event 'onCdAddedToLibrary' and passed in the artist name and title of the track. All plug-ins will receive these parameters, process them and optionally pass back information. You can then handle that information however you like.

In summary, here's the complete example code:

```
JPluginHelper::importPlugin( 'myplugingroup' );
$dispatcher =& JDispatcher::getInstance();
```

```
$results = $dispatcher->trigger( 'onCdAddedToLibrary', array( &$artist
, &$title ) );
```

Note: One thing to notice about the trigger method is that there is nothing defining which group of plug-ins should be notified. In actuality, all plug-ins that have been loaded are notified regardless of the group they are in. So, it's important to make sure you have an event name that does not conflict with any other plug-in group's event name. Most of the time this is not an issue because your component is the one that is loading the plug-in group, so you know which ones are loaded, however be aware that the "system" plugin group is loaded very close to the beginning of the request, so you have to make sure you don't have any event naming conflicts with the system events.