Plugins

Overview

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. 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 CMS functionality.

See System Events for a list of core plugin events.

See Component Events for a list of component plugin events.

Core Types

Plugins are managed at a group level that is defined in the plugin'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 the CMS. These core types are grouped into directories under /plugins. They are:

- antispam
- authentication
- content
- cron
- editors
- editors-xtd
- system
- user

Antispam
: plugins allow you to add to or replace existing anti spam filters to further protect your site against spam and potentially malicious content.

Authentication
: plugins allow you to authenticate (to allow you to login) against different sources. By default you will authenticate against the 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.

Cron
: plugins allow you to add timed "jobs" that are performed at regular intervals. These are good for maintenance tasks, regularly sending emails (i.e., newsletters), etc.

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.

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).

Examples

A plugin demonstrating basic setup:

Download: System Test plugin (.zip)

Structure

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.

Specific plugin files are contained within a directory of the same name as the plugin. While a plugin may contain any number of files and sub-directories, it **must** contain at least two files: the entry point (PHP file of the same name as the plugin) and a XML manifest.

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

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

From the structure detailed above, a "system" plugin called "foo" would have the following file structure:

```
/app
.. /plugins
.. .. /system
.. .. .. /foo
.. .. .. .. foo.php
.. .. .. .. foo.xml
```

There are few restrictions on the file name for the plugin but it is recommended to stick with alpha-numeric characters and underscores only.

Entry Point

Plugins are required to have a file with the same name as the plugin. This is the primary entry

point and will typically contain the plugin class that is to be executed.

Controllers

Overview

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

Structure

Here we have a typical plugin class:

```
<?php
// No direct access
defined( '_HZEXEC_' ) or die();

/**
 * Example system plugin
 */
class plgSystemTest extends HubzeroPluginPlugin
{
 /**
  * Affects constructor behavior.
  * If true, language files will be loaded automatically.
  *
  * @var  boolean
  */
 protected $_autoloadLanguage = false;

 /**
  * Constructor
  *
  * @param  object $subject The object to observe
  * @param  array   $config  An array that holds the plugin configurat
ion
  * @return  void
  */
 public function __construct(&$subject, $config)
 {
  parent::__construct($subject, $config);

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

 /**
```

```
  * Do something onAfterInitialise
  *
  * @return  void
  */
 public 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('_HZEXEC_') 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).

All plugins must extend or be descendants of HubzeroPluginPlugin. 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. This is used only when some work is needed performed when the plugin is actually loaded. This happens with a call to the helper method Plugin::import(*<plugin_type>*). This means that even if the plugin is never triggered, for whatever reason, there is still an opportunity to execute code if needed in the constructor.

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 application sets itself up for work.

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 framework will auto-register all the methods in the class for you.

System 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 CMS, you can include any triggered event in a system plugin.

The events triggered are:

Antispam

- onAntispamDetector
- onAntispamTrain

Authentication

- onAuthenticate

Content

- onContentPrepare
- onAfterDisplayTitle
- onContentBeforeDisplay
- onContentBeforeSave
- onContentAfterSave
- onContentBeforeDelete

Cron

- onCronEvents

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

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.

Views

Overview

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 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
.. .. .. forum.php   (the main plugin file)
.. .. .. forum.xml   (the installation XML file)
.. .. .. /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 HubzeroPluginPlugin
{
    ...
```

```
    public function onReturnHtml()
    {
  // Instantiate a new view
  $view = new HubzeroPluginView(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('_HZEXEC') or die('Restricted access'); // no direct acc
ess ?>
<p>
 <?php echo $this->hello; ?>
</p>
```

Sub-Views

Loading a sub-view (a view within a view, also commonly called a "partial") can now be done via the view() method. This method accepts three arguments: 1) the view name, 2) the parent folder name and 3) the plugin name. If the second and third arguments are not passed, the parent folder is inherited from the view the method is called from (i.e., $this).

Example (called from within a plugin view):

```
... html ...
<?php
 $this->view('layout')
        ->set('foo', $bar)
        ->display();
?>
... html ...
```

Assets

Overview

Although less common than components or modules, sometimes a module to plugin has need for its own styles and scripts to further enhance the user experience. There are a number of helpers to make adding CSS and Javascript to a the document a quick and easy process.

Directory Structure & Files

Assets are stored in the same directory as the plugin file itself and, while there are no hard rules on the placement and organization of the files, it is highly recommended to follow the structure detailed below as it helps keep both small and large projects clean, organized, and allows for several helper methods (detailed in the "Helpers" section).

All assets are stored within an assets folder, which is further sub-divided by asset type. The most common types being js (javascript), css (cascading stylesheets), and img (images) but may also contain any other asset such as fonts, less, and so on.

```
/app
.. /plugins
.. .. /{PluginType}
.. .. .. /{PluginName}
.. .. .. .. /assets
.. .. .. .. .. /css
.. .. .. .. .. /img
.. .. .. .. .. /js
```

Helpers

The HubzeroPluginPlugin class brings with it some useful methods for pushing StyleSheets and JavaScript assets to the document as well as building paths for images. These methods can be called from within the extended helper class or a plugin view.

Cascading Stylesheets

The css() method provides a quick and convenient way to attach stylesheets. It accepts two arguments:

1. The name of the stylesheet to be pushed to the document (file extension is optional). If no name is provided, the name of the plugin will be used. For instance, if called within a view of the members plugin profile, the system will look for a stylesheet named

profile.css.
2. The name of the extension to look for the stylesheet. This accepts either module, component or plugin name and will follow the same naming conventions used for extension directories (e.g. "com_tags", "mod_login", etc). Passing an extension name of "system" will retrieve assets from the core system assets (/core/assets).

For the defined stylesheet to be found, the assets **must** be organized as described in the "Directory Structure & Files" section.

Method chaining is also allowed.

```
<?php
// Push a stylesheet to the document
$this->css()
      ->css('another');
?>
... view HTML ...
```

Javascript

Similarly, a js() method is available for pushing javascript assets to the document. The arguments accepted are the same as the css() method described above.

```
<?php
// Push some javascript to the document
$this->js()
      ->js('another');
?>
... view HTML ...
```

Images

Finally, a img() method is available for building paths to images within the plugin's assets directory. Unlike the css() and js() methods, this helper does not add anything to the global document object and, instead, simply returns an absolute file path.

Given the following directory structure:

```
/app
.. /plugins
.. .. /{PluginType}
```

```
.. .. .. /{PluginName}
.. .. .. .. /assets
.. .. .. .. .. /img
.. .. .. .. .. .. picture.png
```

From a view within the plugin:

```
<!-- Generate the path to the image -->
<img src="<?php echo $this->img('picture.png'); ?>" alt="My picture" /
>
```

Configuration

Overview

Just as with components and modules, each plugin allows for its own set of configuration values that can be set via the administrative interface.

Defining Options

Configuration options can be defined in the plugin's manifest XML file located in the plugin's directory.

```
/app
.. /plugins
.. .. /{PluginType}
.. .. .. /{PluginName}
.. .. .. .. {PluginName}.xml
```

The XML file's root element will have a child node of <config>. Fields are then added and grouped by fieldsets. These fieldsets correspond to the tabs located in the admin side when viewing the plugin's options.

```
<?xml version="1.0" encoding="utf-8"?>
<extension>
 <config>
  <fieldset
   name="greetings"
   label="PLG_HELLO_WORLD_CONFIG_GREETING_SETTINGS_LABEL"
   description="PLG_HELLO_WORLD_CONFIG_GREETING_SETTINGS_DESC"
  >
   <field
    name="greeting"
    type="text"
    label="PLG_HELLO_WORLD_FIELD_GREETING_LABEL"
    description="PLG_HELLO_WORLD_FIELD_GREETING_DESC"
    default=""
   />
  </fieldset>
 </config>
</extension>
```

It is good practice to use the plugin's language file to define all the appropriate strings.

Retrieving Values

One may quickly retrieve the options for any plugin by calling the params() method on the Plugin facade or directly accessing the method on the underlying HubzeroPluginLoader class. This method accepts two arguments of the plugin type and plugin name and returns a HubzeroConfigRegistry object.

```
$params = Plugin::params('hello', 'world');

echo $param->get('greeting');
```

Alternatively, all plugin instances should already have their params available upon instantiation.

```
<?php

class plgHelloWorld extends HubzeroPluginPlugin
{
    public function onGreeting()
    {
        echo $this->params->get('greeting');
    }
}
```

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.

Manifest

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.

```
<?xml version="1.0" encoding="utf-8"?>
<extension version="1.7" type="plugin" group="system">
 <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>
 <config>
  <fieldset>
     <field name="example"
      type="text"
      default=""
      label="Example"
      description="An example text parameter" />
  </fieldset>
 </config>
</extension>
```

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

INSTALL/EXTENSION

This 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/CONFIG

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.

Loading

Triggering Events

Plugins are lazy-loaded by default, which means they must be imported and registered with the event dispatcher on a "as-needed" basis. This can be accomplished by using dot-notation when triggering the event (more on that later) or by manually importing the necessary plugin group:

```
Plugin::import('groups');
```

The above line will import all published plugins of the type "groups"—that is, all plugins in /plugins/groups—and register them as event listeners with the dispatcher.

To fire an event, one may use the Event facade, passing an instance of the event to the trigger method. The trigger method will dispatch the event to all of its registered listeners:

```
// Import the "media" plugins
Plugin::import('media');

// Trigger the event
$results = Event::trigger('onAlbumAddedToLibrary', array($artist, $tit
le));
```

Here we have triggered the event 'onAlbumAddedToLibrary' and passed in the artist name and title of the album. All plug-ins will receive these parameters, process them and optionally pass back information. The trigger method will always return an array.

Although relatively short, the above code example can be simplified even further by using dot-notation to combine the plugin group and event name into one:

```
// Load the plugin group "media" and trigger the event
$results = Event::trigger('media.onAlbumAddedToLibrary', array($artist
, $title));
```

Here, the trigger method recognizes dot-notation being used and extracts the plugin group from the string, imports said plugin group, and registers them with the event dispatcher before

triggering the event. For those concerned about performance, it should be noted the importing of plugins *will only happen once*.

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 be sure that event names do not conflict with any other plugin group's event name.

Stopping an Event

Sometimes, a plugin may need to prevent any further plugins from responding to an event. In such cases, the event loop can be halted.

When an event is triggered, an event object is created to track responders, pass data, and collect responses from listeners. For anonymous functions, this event object is passed as the only argument. For legacy plugins, the object is attached as a public property to the plugin and can be accessed by calling $this->event.

So, stopping an event is done by calling stop on the event object.

```
<?php
class plgSystemExample extends Plugin
{
    public function onAfterRoute()
    {
        // ... some logic here ...

        $this->event->stop();
    }
}
```