Modules

Overview

Modules are extensions which present certain pieces of information on your site. It's a way of presenting information that is already present. This can add a new function to an application which was already part of your website. Think about latest article modules, login module, a menu, etc. Typically you'll have a number of modules on each web page. The difference between a module and a module is not always very clear for everybody. A module doesn't make sense as a standalone application, it will just present information or add a function to an existing application. Take a newsletter for instance. A newsletter is a module. You can have a website which is used as a newsletter only. That makes perfectly sense. Although a newsletter module probably will have a subscription page integrated, you might want to add a subscription module on a sidebar on every page of your website. You can put this subscribe module anywhere on your site.

Another commonly used module would be a search box you wish to be present throughout your site. This is a small piece of re-usable HTML that can be placed anywhere you like and in different locations on a template-by-template basis. This allows one site to have the module in the top left of their template, for instance, and another site to have it in the right side-bar.

Directory Structure & Files

The directory structure used allows you to separate different MVC applications into self-contained units. This helps keep related code organized, easy to find, and can make redistribution as packages considerably easier. To illustrate the typical module directory structure and files:

```
/hubzero
  /modules
    /mod_{ModuleName}
      /tmpl
        default.php
      helper.php
      mod_{ModuleName}.php
      mod_{ModuleName}.xml
```

A Joomla! 1.5 Module is in its most basic form two files: an XML configuration file and a PHP controller file. Typically, however, a module will also include a view file which contains the HTML and presentation aspects.

/tmpl

This directory contains template files.

default.php

This is the module template. This file will take the data collected by mod_{ModuleName}.php and generate the HTML to be displayed on the page.

helper.php

This file contains the helper class which is used to do the actual work in retrieving the information to be displayed in the module (usually from the database or some other source).

mod_{ModuleName}.php

This file is the main entry point for the module. It will perform any necessary initialization routines, call helper routines to collect any necessary data, and include the template which will display the module output.

mod_{ModuleName}.xml

The XML configuration file contains general information about the module (as will be displayed in the Module Manager in the Joomla! administration interface), as well as module parameters which may be supplied to fine tune the appearance / functionality of the module.

While there is no restriction on the name itself, all modules must be prefixed with "mod_".

Examples

A simple "Hello, World" module:

Download: Hello World module (.zip)

A module demonstrating database access and language file:

Download: List Names module (.zip)

Installation

Installing

See Installing Extensions for details.

Uninstalling

See Uninstalling Extensions for details.

Manifests

Overview

All modules should include a manifest in the form of an XML document named the same as the module. The file holds key "metadata" about the module.

Note: All modules must be prefixed with mod_.

Directory Structure & Files

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

```
/hubzero
  /modules
    /{ModuleName}
      /tmpl
        default.php
      helper.php
      mod_{ModuleName}.php
      mod_{ModuleName}.xml
```

Structure

This XML file just lines out basic information about the template such as the owner, version, etc. for identification by the Joomla! installer and then provides optional parameters which may be set in the Module Manager and accessed from within the module's logic to fine tune its behavior. Additionally, this file tells the installer which files should be copied and installed.

A typical module manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<install type="module" version="1.5.0">
 <!-- Name of the Module -->
 <name>Hello World - Hello</name>

 <!-- Name of the Author -->
 <author>Ambitionality Software LLC</author>

 <!-- Version Date of the Module -->
 <creationDate>2008-06-23</creationDate>
 <!-- Copyright information -->
 <copyright>All rights reserved by Ambitionality Software LLC 2008.</c
opyright>

 <!-- License Information -->
 <license>GPL 2.0</license>

 <!-- Author's email address -->
 <authorEmail>info@ambitionality.com</authorEmail>

 <!-- Author's website -->
 <authorUrl>www.ambitionality.com</authorUrl>

 <!-- Module version number -->
 <version>1.0.0</version>

 <!-- Description of what the module does -->
 <description>Outputs a random list of user names</description>

 <!-- Listing of all files that should be installed for the module to
function -->
 <files>
  <!-- The "module" attribute signifies that this is the main controll
er file -->
  <filename module="mod_listnames">mod_listnames.php</filename>
  <filename>index.html</filename>
  <filename>tmpl/default.php</filename>
  <filename>tmpl/index.html</filename>
 </files>

 <languages>
  <!-- Any language files included with the module -->
  <language tag="en-GB">en-GB.mod_listnames.ini</language>
 </languages>

 <!-- Optional parameters -->
 <params>
  <!-- parameter to allow placement of a module class suffix for the m
odule table / xhtml display -->
  <param name="moduleclass_sfx" type="text" default="" label="Module C
lass Suffix" description="PARAMMODULECLASSSUFFIX" />

  <!-- just gives us a little room between the previous paramter and t
he next -->
  <param name="@spacer" type="spacer" default="" label="" description=
"" />

  <!-- A parameter that allows an administrator to modify the number o
f users that this module will display -->
  <param name="usercount" type="text" default="5" label="LABEL USER CO
UNT" description="DESC USER COUNT" />
 </params>
</install>
```

Note: Notice that we DO NOT include a reference in the files section for the XML file.

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

INSTALL
: The install tag has several key attributes. The type must be "module".

NAME
: You can name the module in any way you wish.

FILES
: The files tag includes all of the files that will will be installed with the module.

PARAMS
: Any number of parameters can be specified for a module.

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

Controllers

Overview

Unlike components, which potentially can have multiple controllers, modules do not have a controller class. As such, the module directory structure doesn't include a /controllers subdirectory or controller.php. Instead, the setting of parameters, inclusion of any necessary files, and the instantiation of the module's view are done within the mod_{ModuleName}.php file.

Directory Structure & Files

The controller is stored in the same directory as the module file itself and must be named the same (the file extension being the obvious difference).

```
/hubzero
  /modules
    /{ModuleName}
      /tmpl
        default.php
      helper.php
      mod_{ModuleName}.php
      mod_{ModuleName}.xml
```

Implementation

Most modules will perform three tasks in the following order:

- Include the helper.php file which contains the class to be used to collect any necessary data
- Invoke the appropriate helper class method to retrieve any data that needs to be available to the view
- Include the template to display the output

Here are the contents of mod_listnames.php:

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

// Include the helper file
require_once(dirname(__FILE__).DS.'helper.php');
```

```
// Get a parameter from the module's configuration
$userCount = $params->get('usercount');

// Get the items to display from the helper
$items = modListNamesHelper::getItems($userCount);

// Include the template for display
require(JModuleHelper::getLayoutPath('mod_listnames'));
```

Helpers

Overview

The helper.php file contains that helper class that is used to retrieve the data to be displayed in the module output. Most modules will have at least one helper but it is possible to have a module with more or none.

Directory Structure & Files

The directory structure used for MVC oriented modules includes the helper.php file in the top directory for that module. While there is no rule stating that we must name our helper class as we have, but it is helpful to do this so that it is easily identifiable and locateable.

```
/hubzero
  /modules
    /mod_{ModuleName}
      helper.php
```

Implementation

In our mod_helloworld example, the helper class will have one method: getItems(). This method will return the items we retrieved from the database.

Here is the code for the mod_helloworld helper.php file:

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

class modHelloWorldHelper
{
    /**
     * Retrieves the hello message
     *
     * @param array $params An object containing the module parameters
     * @access public
     */
    public function getItems( $userCount )
    {
        return 'Hello, World!';
    }
}
```

More advanced modules might include multiple database requests or other functionality in the helper class method.

Languages

Setup

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

```
; Module - List Names (en-US)
MOD_LISTNAMES_LABEL_USER_COUNT = "User Count"
MOD_LISTNAMES_DESC_USER_COUNT = "The number of users to display"
MOD_LISTNAMES_RANDOM_USERS = "Random Users for Hello World"
MOD_LISTNAMES_USER_LABEL = "%s is a randomly selected user"
```

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 MOD_{ModuleName}_{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.

Translating Text

Below is an example of accessing the translate helper:

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

JText::_ is used for simple strings.
JText::sprintf is used for strings that require dynamic data passed to them for variable replacement.

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

See the Languages overview for details.

Layouts

Overview

While technically not necessary for a module to function, it is considered best practices to have a more MVC structure to your module 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 module and component presentation in templates is further explained in the Templates: Overrides section.

Directory Structure & Files

The directory structure used for MVC oriented modules includes a tmpl directory for storing view files. While more views may be possible, modules should include at least one view names default.php.

```
/hubzero
  /modules
    /mod_{ModuleName}
      /tmpl
        default.php
```

Implementation

A simple view (default.php) for a module named mod_listnames:

```
<?php defined('_JEXEC') or die('Restricted access'); // no direct acce
ss ?>
<?php echo JText::_('MOD_LISTNAMES_RANDOM_USERS'); ?>
<ul>
 <?php foreach ($items as $item) { ?>
 <li>
  <?php echo JText::sprintf('MOD_LISTNAMES_USER_LABEL', $item->name);
?>
 </li>
 <?php } ?>
</ul>
```

Here we simply create an unordered HTML list and then iterate through the items returned by our helper (in mod_listnames.php), printing out a message with each user's name.

An important point to note is that the template file has the same scope as the mod_listnames.php file. What this means is that the variable $items can be defined in the mod_listnames.php file and then used in the default.php file without any extra declarations or function calls.

Now that we have a view to display our data, we need to tell the module to load it. This is done in the module's controller file and typically occurs last.

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

// Include the helper file
require_once(dirname(__FILE__).DS.'helper.php');

// Get a parameter from the module's configuration
$userCount = $params->get('usercount');

// Get the items to display from the helper
$items = modListNamesHelper::getItems($userCount);

// Include the template for display
require(JModuleHelper::getLayoutPath('mod_listnames'));
```

Here we can see that the name of the module must be passed to the getLayoutPath method of JModuleHelper. This will load default.php and stores the output in an output buffer which is then rendered onto the page output.

Packaging

Overview

Packaging a module for distribution is easy. Just "zip" up the module directory into a compressed archive file. When the ZIP file is installed, the language file is copied to /language/{LanguageName}/{LanguageName}.{ModuleName}.ini and is loaded each time the module is loaded. All of the other files are copied to the /modules/{ModuleName} subfolder of the Joomla! installation.

Loading

Loading in Templates

Modules may be loaded in a template by including a Joomla! specific jdoc:include tag. This tag includes two attributes: type, which must be specified as module in this case and name, which specifies the position that you wish to load. Any modules assigned to the specified position (set via the administrative Module Manager) declared in the name attribute will have their output placed in the template (the jdoc:include is removed by Joomla! afterwards).

```
<jdoc:include type="modules" name="footer" />
```

Advanced Template Loading

The countModules method can be used within a template to determine the number of modules enabled in a given module position. This is commonly used to include HTML around modules in a certain position only if at least one module is enabled for that position. This prevents empty regions from being defined in the template output and is a technique sometimes referred to as "collapsing columns".

For example, the following code includes modules in the 'user1' position only if at least one module is enabled for that position.

```
<?php if ($this->countModules( 'user1' )) : ?>
 <div class="user1">
  <jdoc:include type="modules" name="user1" />
 </div>
<?php endif; ?>
```

The countModules method can be used to determine the number of Modules in more than one Module position. More advanced calculations can also be performed.

The argument to the countModules function is normally just the name of a single Module position. The function will return the number of Modules currently enabled for that Module position. But you can also do simple logical and arithmetic operations on two or more Module positions.

```
$this->countModules( 'user1 + user2' );
```

Although the usual arithmetic operators, +. -. *, / will work as expected, these are not as useful as the logical operators 'and' and 'or'. For example, to determine if the 'user1' position and the 'user2' position both have at least one Module enabled, you can use the function call:

```
$this->countModules( 'user1 and user2' );
```

Careful: A common mistake is to try something like this:

```
$this->countModules( 'user1' and 'user2' );
```

This will return false regardless of the number of Modules enabled in either position, so check what you are passing to countModules carefully.

You must have exactly one space character separating each item in the string. For example, 'user1+user2' will not produce the desired result as there must be a space character either side of the '+' sign. Also, 'user1 &nbp;+ user2' will produce an error message as there is more than one space separating each element.

Example using the or operator: The user1 and user2 Module positions are to be displayed in the region, but you want the region to not appear at all if no Modules are enabled in either position.

```
<?php if ($this->countModules( 'user1 or user2' )) : ?>
 <div class="rightcolumn">
  <jdoc:include type="modules" name="user1" />
  <jdoc:include type="modules" name="user2" />
 </div>
<?php endif; ?>
```

Advanced example: The user1 and user2 Module positions are to be displayed side-by-side with a separator between them. However, if only one of the Module positions has any Modules enabled then the separator is not needed. Furthermore, if neither user1 or user2 has any Modules enabled then nothing is output.

```
<?php if ($this->countModules( 'user1 or user2' )) : ?>
 <div class="user1user2">

 <?php if ($this->countModules( 'user1' )) : ?>
```

```
  <jdoc:include type="modules" name="user1" style="xhtml" />
 <?php endif; ?>

 <?php if ($this->countModules( 'user1 and user2' )) : ?>
  <div class="greyline"></div>
 <?php endif; ?>

 <?php if ($this->countModules( 'user2' )) : ?>
  <jdoc:include type="modules" name="user2" style="xhtml" />
 <?php endif; ?>

 </div>
<?php endif; ?>
```

Notice how the first countModules call determines if there any Modules to display at all. The second determines if there are any in the 'user1' position and if there are it displays them. The third call determines if both 'user1' and 'user2' positions have any Modules enabled and if they do then if provides a separator between them. Finally, the fourth call determines if there are any enabled Modules in the 'user2' position and displays them if there are any.

Loading in Components

Sometimes it is necessary to render a module within a component. This can be done with the XModuleHelper class provided by HUBzero. To import the class, you must first use the ximport('name of file or class') method.

XModuleHelper::renderModules($position)

> Used for loading potentially multiple modules assigned to a position. This will capture the rendered output of all modules assigned to the $position parameter passed to it and return the compiled output.
>
> ```
> ximport('xmodule');
> $output = XModuleHelper::renderModules('footer');
> ```

XModuleHelper::renderModule($name)

> Used for loading a single module of a specific name. This will capture the rendered output of the module with the $name parameter passed to it and return the compiled

output.

```
ximport('xmodule');
$output = XModuleHelper::renderModule('mod_footer');
```

XModuleHelper::displayModules($position)

Used for loading a single module of a specific name. This will echo rendered output of the module with the $name parameter passed to it.

```
ximport('xmodule');
XModuleHelper::displayModules('footer');
```

XModuleHelper::renderModule($name)

Used for loading a single module of a specific name. This will output the module with the $name parameter passed to it.

```
ximport('xmodule');
XModuleHelper::displayModule('mod_footer');
```

Loading in Articles

Modules may be loaded in an article by including a specific {xhub:module} tag. This tag includes one required attribute: position, which specifies the position that you wish to load. Any modules assigned to the specified position (set via the administrative Module Manager) declared in the position attribute will have their output placed in the article in the location of the {xhub:module} tag.

```
{xhub:module position="footer"}
```

Note: To use this feature, the xHUB Tags plugin for content must be installed and active.