Modules

Overview

Modules are lightweight and flexible extensions used for page rendering. These modules are often “boxes” arranged around a component on a typical page. Some modules are linked to components, displaying information specific to or feeding information to that component. An example of this would be a "Report a problem" module that presents a form on every page for creating a ticket in the support component. However, modules do not need to be linked to components; they can be just static HTML or text.

Modules are meant to be small pieces of re-usable HTML that can be placed anywhere desired 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.

Examples

A simple "Hello, World" module:

Download: Hello World module (.zip)

A module demonstrating database access and language file:

Download: List Names module (.zip)

Structure

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:

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

A 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 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_".

Implementation

Most modules will perform three tasks in the following order:

- Define the module namespace
- Include the helper.php file which contains the class to be used to collect any necessary data and render it
- Instantiate the helper class and call the display() method which will:
    - 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
- Include the template to display the output

Here are the contents of mod_listnames.php:

```
<?php
// Define the namespace
namespace ModulesListNames;

// Include the helper file
require_once __DIR__ . DS . 'helper.php';

// Instantiate the module helper and call its display() method
with(new Helper($params, $module))->display();
```

Helpers

Overview

Unlike components, which potentially can have multiple controllers, modules do not require 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 helper.php file.

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.

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

Implementation

In our mod_helloworld example, the helper class will have one method: display(). This method will output the contents of the module.

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

```
<?php
namespace ModulesHelloWorld;

use HubzeroModuleModule;

class Helper extends Module
{
    public function display()
    {
        echo 'Hello, World!';
    }
```

```
}
```

More advanced modules might include multiple database requests or other functionality in the helper class method, passing data to a view and rendering the view.

```
<?php
namespace ModulesHelloWorld;

use HubzeroModuleModule;
use App;

class Helper extends Module
{
    public function display()
    {
        // Retrieve rows from the database
        $this->rows = $this->getItems();

        // Render the view
        require $this->getLayoutPath();
    }

    public function getItems()
    {
        $db = App::get('db');
        $db->setQuery(" ... ");
        return $db->loadObjectList();
    }
}
```

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 Lang::txt("MOD_EXAMPLE_MY_LINE"); ?></p>
```

Lang::txt() is used for both simple strings and 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.

Views

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 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.

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

Implementation

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

```
<?php defined('_HZEXEC_') or die(); // no direct access ?>
<?php echo Lang::txt('MOD_LISTNAMES_RANDOM_USERS'); ?>
<ul>
 <?php foreach ($this->items as $item) : ?>
 <li>
  <?php echo Lang::txt('MOD_LISTNAMES_USER_LABEL', $item->name); ?>
 </li>
 <?php endforeach; ?>
</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 display() method. What this means is that the variable $items can be defined in the helper.php file, assigned to $this 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('_HZEXEC_') or die();

class modHelloWorld extends HubzeroModuleModule
{
    /**
     * Retrieves the hello message
     *
     * @param array $params An object containing the module parameters
     * @access public
     */
    public function display()
    {
        $this->greeting = 'Hello, World!';

        parent::display();
    }
}
```

Here we can see that display() method calls its parent class' display() method which, in turn loads the module's view. This will load default.php and stores the output in an output buffer which is then rendered onto the page output.

Assets

Overview

It is not uncommon for a module to have 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 module a quick and easy process.

Directory Structure & Files

Assets are stored in the same directory as the module 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
.. /modules
.. .. /{ModuleName}
.. .. .. /assets
.. .. .. .. /css
.. .. .. .. /img
.. .. .. .. /js
```

Helpers

The HubzeroModuleModule class brings with it some useful methods for pushing StyleSheets and JavaScript assets to the document. These methods can be called from within the extended helper class or the view itself.

Cascading Stylesheets

The css() method provides a quick and convenient way to attach stylesheets. For modules, 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 module will be used. For instance, if called within a view of the module mod_tags, the system will look for a stylesheet named mod_tags.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 module'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
.. /modules
.. .. /{ModuleName}
.. .. .. /assets
```

```
.. .. .. .. /img
.. .. .. .. .. picture.png
```

From a component view:

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

Packaging

Overview

It is possible to install a module manually by copying the files using an SFTP client and modifying the database tables. It is more efficient to create a package file in the form on a composer.json document that will allow the Installer to do this for you. This package file resides in the top-level of your module's directory and contains a variety of information:

- basic descriptive details about your module (i.e. name), and optionally, a description, copyright and license information.
- the extension type (component, module, plugin, template)
- optionally, a destined install directory

Composer Manifest

This composer.json file just outlines basic information about the module such as the owner, version, etc. for identification by the installer and then tells the installer which files should be copied and installed.

A typical component manifest:

```
{
  "name": "myorg/mod_example",
  "description": "Example module",
  "license": "MIT",
  "type": "hubzero-module"
}
```

The hub includes some extra code that tells Composer where/how to install extensions, so it's important to use the designated types. Available types are: hubzero-component, hubzero-module, hubzero-plugin, hubzero-template.

Structure

Packaging a module for distribution is relatively easy. The file and directory structure is exactly as it would be after installation. Here's what a typical package will look like:

```
/mod_{name}
```

```
  mod_{name}.xml
  mod_{name}.php
  composer.json
  /tmpl
    default.php
```

XML Manifest (deprecated)

All modules should include a manifest in the form of an XML document named the same as the module. This file lines out basic information about the module such as the owner, version, etc. for identification by the 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"?>
<extension type="module" version="1.0.0">
 <!-- Name of the Module -->
 <name>mod_listnames</name>

 <!-- Name of the Author -->
 <author>HUBzero</author>

 <!-- Version Date of the Module -->
 <creationDate>2015-06-23</creationDate>

 <!-- Copyright information -->
 <copyright>All rights reserved by HUBzero 2015.</copyright>

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

 <!-- Author's email address -->
 <authorEmail>support@hubzero.org</authorEmail>

 <!-- Author's website -->
 <authorUrl>hubzero.org</authorUrl>

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

 <!-- Description of what the module does -->
```

```
 <description>MOD_LISTNAMES_DESCRIPTION</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>helper.php</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 -->
 <config>
  <fields name="params">
   <fieldsset name="basic">
    <!-- parameter to allow placement of a module class suffix for the
 module table / xhtml display -->
    <field name="moduleclass_sfx" type="text" default="" label="MOD_LI
STNAMES_PARAM_CLASS_LABEL" description="MOD_LISTNAMES_PARAM_CLASS_DESC
" />

    <!-- just gives us a little room between the previous paramter and
 the next -->
    <field name="@spacer" type="spacer" default="" label="" descriptio
n="" />

    <!-- A parameter that allows an administrator to modify the number
 of users that this module will display -->
    <field name="usercount" type="text" default="5" label="MOD_LISTNAM
ES_PARAM_USERCOUNT_LABEL" description="MOD_LISTNAMES_PARAM_USERCOUNT_D
ESC" />
   </fieldset>
  </fields>
 <config>
</extension>
```

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:

EXTENSION
: The extension 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.

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

Loading

Loading in Templates

Modules may be loaded in a template by including a 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 the CMS 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="container">
  <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" />
 <?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" />
 <?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 HubzeroModuleHelper class provided by HUBzero.

HubzeroModuleHelper::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.
>
> ```
> $output = HubzeroModuleHelper::renderModules('footer');
> ```

HubzeroModuleHelper::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.

```
$output = HubzeroModuleHelper::renderModule('mod_footer');
```

HubzeroModuleHelper::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.

```
HubzeroModuleHelper::displayModules('footer');
```

HubzeroModuleHelper::renderModule($name)

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

```
HubzeroModuleHelper::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.

Migrations

All the common extension types for HUBzero can include their own migrations directory. Migrations are used for installing the extension into the required tables for the CMS to know about said extension's existence, installing any needed tables, installing sample data, etc.

To illustrate the typical component directory structures and files:

```
/app
.. /modules
.. .. /mod_example
.. .. .. /migrations
.. .. .. .. /Migration20190301102219ModExample.php
.. .. .. /tmpl
.. .. .. .. default.php
.. .. .. helper.php
.. .. .. mod_example.php
.. .. .. mod_example.xml
```

See the Migrations documentation for more about naming conventions, setup, etc.

Modules typically have at least one initial migration for registering the extension with the CMS. This migration typically just involves calling the addModuleEntry helper method:

```
<?php

use HubzeroContentMigrationBase;

// No direct access
defined('_HZEXEC_') or die();

/**
 * Migration script for registering the example plugin
 **/
class Migration20190301102219ModExample extends Base
{
    /**
     * Up
     **/
    public function up()
    {
        // Register the module
```

```
        //
        // @param   string  $element  (required) Module element
        // @param   int     $enabled  (optional, default: 1) Whether o
r not the module should be enabled
 // @param   string  $params          (optional) Plugin params (if alr
eady known)
 // @param   int     $client   (optional, default: 0) Client [site=0,
admin=1]
        $this->addModuleEntry('mod_example');
    }

    /**
     * Down
     **/
    public function down()
    {
        $this->deleteModuleEntry('mod_example');
    }
}
```

That's all there is to it! The addModuleEntry method adds the necessary entries to the needed database tables for the CMS to be aware of the module's existence.

For a module to appear anywhere, a new instance of that module must be created and assigned to a position. This above migration just makes the CMS aware of the module's presence so that a new instance *can* be created.