Web Developers

Written in a book format, it contains the information a developer needs to not only understand and use HUBzero components but build extensions for a HUBzero installation. Developers will learn how to use common objects, available code libraries and utilities, and distinguish between and develop the following kinds of extensions:

- Components
- Modules
- Plugins
- Templates

Introduction

Getting Started

As a developer you are tasked with altering or extending the functionality of a HUBzero install or one of its extensions. You will need to be proficient in PHP and have some familiarity with such things as JavaScript or CSS. If you are new to HUBzero, this reference should help guide you through the creation of extensions such as modules and widgets (more on those later). Thankfully, the requirements for getting started creating HUBzero extensions are minimal: knowledge of programming in PHP and a good text editor. While those are the only *requirements* we do, however, recommend you have working knowledge of the following:

- HTML
- Cascading StyleSheets (CSS)
- JavaScript (familiarity with the [jQuery](#) framework is a plus)
- XML
- Model-View-Controller (MVC) design pattern
- Object-Oriented Programming

Upgrade Guide

Directory Structure & Files

Most notable about the 2.0.0 release will be the new directory structure and reorganization of the various files and extensions comprising the CMS.

Files are essentially divided between two primary directories: app and core.

```
/app
/core
index.php
```

The app directory is where everything concerning a specific hub lives. That is, it's the home to all the logs, cache data, uploads, and extensions unique to a specific instance of a hub.

Constants

Joomla	Hubzero
JPATH_ROOT	PATH_ROOT
JPATH_BASE	PATH_ROOT
JPATH_SITE	PATH_ROOT
JPATH_ADMINISTRATOR	PATH_ROOT
	No files or code should be placed into or read from the administrator directory and it is slated for deletion in a future version.
JPATH_COMPONENT	Component::path($option)
n/a	PATH_APP
	Points to ROOT/app where all hub-specific data resides.
n/a	PATH_CORE
	Points to ROOT/core where the framework and core extensions live.
_JEXEC	_HZEXEC_

It is highly recommended, when including files within the same extension (component, module, plugin), to use the __DIR__ and __FILE__ PHP constants and relative paths.

```php
<?php
// This file is example.php, located in:
// ROOT/app/components/com_example/admin
```

```
// dirname(__DIR__) moves up one directory
// ROOT/app/components/com_example/models
require_once(dirname(__DIR__) . DS . 'models' . DS . 'foo.php');

// ROOT/app/components/com_example/admin/controllers
require_once(__DIR__ . DS . 'controllers' . DS . 'example.php');
```

Common Classes

To make upgrading an extension a little easier, a number of Joomla classes (and their methods) have equivalent classes in the new framework.

JRoute

JRoute::_();   Route::url();

JText

The JText class, used for translating language keys, was replaced by the Lang facade. Along with this, the _() and sprintf() methods were merged to allow for a single call to Lang::txt() with a variable number of arguments. If more than one argument is passed to the txt() method, the translator will attempt to perform variable replacement in the translated string.

```
// Language file
COM_EXAMPLE_HELLO="Hello!"
COM_EXAMPLE_HELLO_NAME="Hello, %s!"

...

// PHP

// Outputs 'Hello!'
echo Lang::txt('COM_EXAMPLE_HELLO');

// Outputs 'Hello, HUBzero!'
echo Lang::txt('COM_EXAMPLE_HELLO_NAME', 'HUBzero');
```

JText::_();   Lang::txt();
JText::sprintf();   Lang::txt();
JText::plural();   Lang::txts();
JText::alt();   Lang::alt();

JRequest

To make transitioning easier, all public JRequest methods have been preserved on the global request object, which can be accessed through the application container or the Request facade.

```
// Via the application container
$request = App::get('request');
$foo = $request->getVar('foo');

// Via the facade
$foo = Request::getVar('foo');
```

In the majority of cases, this means simply dropping the 'J' from JRequest will be sufficient for upgrading an extension's code.

JRequest::*   Request::*

JToolbarHelper

Perhaps one of the easier conversions; Simply replace instances of JToolbarHelper with the Toolbar facade. Method names and the arguments passed to them stay the same.

```
// Joomla
JToolbarHelper::publishList();
JToolbarHelper::unpublishList();

// Hubzero
Toolbar::publishList();
Toolbar::unpublishList();
```

JSubMenuHelper

As with JToolbarHelper above, only the class name need be updated. All primary method names stay the same.

```
// Joomla
JSubMenuHelper::addEntry(
 JText::_('COM_COLLECTIONS_POSTS'),
 'index.php?option=com_collections&controller=posts',
 $controllerName == 'posts'
);
```

```
// Hubzero
Submenu::addEntry(
 Lang::txt('COM_COLLECTIONS_POSTS'),
 Route::url('index.php?option=com_collections&controller=posts'),
 $controllerName == 'posts'
);
```

JHtml

Unlike may of the other classes mentioned above, the class, method, and arguments changed for the replacement of Html. Easily enough, the "J" can simply be dropped to have a class name of just Html. The method name and first argument passed to said method is a little more complicated but follows a strict pattern. For Html, all arguments were passed to a method of _(), the first argument being a dot-notation combination of sub library and the function to call within it.

```
echo JHTML::_('grid.sort', 'COM_COLLECTIONS_COL_TITLE', 'title', @$this->filters['sort_Dir'], @$this->filters['sort']);
```

For the Html class, the method is now the name of the sub-library and the first argument passed is the name of the function to call.

```
echo Html::grid('sort', 'COM_COLLECTIONS_COL_TITLE', 'title', @$this->filters['sort_Dir'], @$this->filters['sort']);
```

Examples:

```
// Joomla
JHtml::_('behavior.framework');

// Hubzero
Html::behavior('framework');
```

Factory Objects

The following is a list of conversions for objects typically acquired from Joomla's JFactory. In most cases, the objects or their equivalents are available for retrieval from the global App. A number of the objects also have associated Facades for quicker access. In the examples below **method()** is variable and implies that the method formerly called on the Joomla object can be called statically on the facade.

Example 1:

```
// Joomla
$user = JFactory::getUser();
echo $user->get('name');


// Hubzero
echo User::get('name');
```

Example 2:

```
// Joomla
$doc = JFactory::getDocument();
$doc->addStylesheet('/some/file.css');

// Hubzero
Document::addStylesheet('/some/file.css');
```

Joomla	Hubzero	Hubzero Facade
JFactory::getDbo();	App::get('db');	n/a
JFactory::getUser();	User::getRoot();	User::**method()**;
JFactory::getSession();	App::get('session');	Session::**method()**;
JFactory::getDocument();	App::get('document');	Document::**method()**;
JFactory::getConfig();	App::get('config');	Config::**method()**;
JFactory::getLanguage();	App::get('lang');	Lang::**method()**;
JFactory::getCache();	App::get('cache.store');	Cache::**method()**;
JFactory::getLogger();	App::get('log.debug');	Log::**method()**;

Dates

Along with a replacement class for Joomla's JDate, the CMS includes a global Date class to make handling and formatting of dates a little easier.

Now

The Date class will always return an instance of HubzeroUtilityDate. If no specific time or timestamp is specified, it will default to 'now'.

```
// Output the current timestamp (UTC) in the database's format. ex: "2
015-04-03 12:23:56"
echo Date::toSql();

// Output the current timestamp (UTC) in Unix format.
echo Date::toUnix();

// Output the current timestamp (UTC) year. ex: "2015"
echo Date::format('Y');

// Output the current timestamp adjusted to the timezone of the hub. F
or example, if the UTC time is "12:23 pm" and the hub's set timezone i
s Eastern Standard Time (EST), the time outoutted will be "08:23 am"
echo Date::toLocal('g:i a');
```

Specified date

A specific timestamp and timezone can be passed to the of method. If no timezone is provided, the timezone will default to UTC.

```
// Output the current timestamp (UTC) year. ex: "2013"
echo Date::of('2013-08-12 17:01:34')->format('Y');

// Output the current timestamp adjusted to the timezone of the hub. e
x: "1:01 pm"
echo Date::of('2013-08-12 17:01:34')->toLocal('g:i a');
```

Users

The global user object, retrieved from JFactory::getUser() can now be accessed anywhere within the CMS from the User facade. Any method, other than getInstance(), statically called on User will be acted upon the current, global user. This is the same as calling JFactory::getUser()->method().

```
// Joomla
echo JFactory::getUser()->get('name');

// Hubzero
echo User::get('name');
```

The getRoot() method can be used to retrieve the underlying object (of the facade) and assigned to a variable as needed.

```
// Joomla
$user = JFactory::getUser();

// Hubzero
$user = User::getRoot();
```

Obtaining instances of new users can be achieved by calling getInstance($id_or_username) on the User facade in the same manner as calling JUser::getInstance($id_or_username) or JFactory::getUser($id_or_username).

```
// Joomla
$user = JFactory::getUser(1234);
// ... or ...
$user = JUser::getInstance(1234);

// Hubzero
$user = User::getInstance(1234);
```

Contribution Guide

Bug Fixes & Patches

The HUBzero Foundation [accepts bug fixes and core patches](). Submissions are reviewed by Foundation technical staff and, if accepted, they will become part of the HUBzero core distribution. When contributions are offered to the HUBzero Foundation, the copyright for the software must be reassigned to the HUBzero Foundation so that the changes can be managed as an indistinguishable part of the core distribution. The HUBzero Foundation reserves the right to reject submissions for any reason.

Adherence to Standards

The HUBzero Foundation has established [coding conventions and guidelines]() for developing HUBzero components. All submissions to the core must adhere to the guidelines, coding patterns, and styles laid forth in the documentation. Contributions are reviewed by the Foundation team before being incorporated into the core distribution and those that do not meet the standards may be tweaked or rewritten as needed, or may be accepted pending required changes. Any contribution known to contain security vulnerabilities may be rejected entirely.

Hosted Hubs

All code extensions and alterations must adhere to the aforementioned coding conventions and guidelines for hubs hosted and maintained by HUBzero. Third-party extensions are allowable but must first be reviewed and accepted to ensure compatibility, functionality, and security concerns are addressed.

Guidelines

- Be sure that the code follows the [development styles and conventions]()
- Make sure it works.
- If it requires internationalization, documentation, or help to be written, be sure you've done that before submission.
- Variables, subroutines, and comments in the code should be made in English. While the talents of the worldwide community is greatly appreciated, we cannot support code that we cannot read.

Subjective Requirements

Below are some subjective requirements that should be taken into consideration.

1. **Is this feature useful?**
   Consider whether this feature is useful to the current or future users. If there are relatively few people for who this is useful: what other reasons are there to include this? If this feature is useful only to some, can it be made in a way that it can be switched off

or easily removed, so that it's out of the way of the average user.

2. **Is this feature usable by the target audience?**
Consider who is the target audience? Is it user-friendly for regular users, does a content manager in an avarage business environment have the skills to use it? Or is this directed to system administrators?

3. **What are the maintenance costs of it?**
Strongly consider the time and effort spent in maintaining it.

4. **Does this fit in the direction HUBzero is going?**
This may be something you want to discuss with other hub owners and users, but you can also submit inquiries to the HUBzero team.

5. **Is there already a similar feature?**
If so: could this similar feature be adapted to suit you needs? Or does your contribution have all the features of the existing feature? And if so, is there a way to upgrade?

Style Guide

Overview

This document provides guidelines for code formatting and documentation to individuals and teams contributing to HUBzero CMS.

Topics covered:

- PHP File Formatting
- PHP and Database Naming Conventions
- PHP, CSS Coding Style
- PHP Inline Documentation

PHP Coding Styles

Code Demarcation

PHP code must always be delimited by the full-form, standard PHP tags:

```
<?php

?>
```

Short tags are never allowed.

For files that contain only PHP code, the closing tag ("?>") is never permitted. It is not required by PHP, and omitting it prevents the accidental injection of trailing white space into the response.

Indention

Indentation should consist of 1 tab per indentation level. Spaces are not allowed.

Line Length

The target line length is 120 characters. Longer lines are acceptable as long as readability is maintained.

Line Termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character. Linefeed characters are represented as ordinal 10, or hexadecimal 0×0A.

Note: Do not use carriage returns (CR) as is the convention in Apple OS's (0×0D) or the carriage return – linefeed combination (CRLF) as is standard for the Windows OS (0×0D, 0×0A).

Strings

String Literals

When a string is literal (contains no variable substitutions), the apostrophe or "single quote" should always be used to demarcate the string:

```
$a = 'Example String';
```

String Literals Containing Apostrophes

When a literal string itself contains apostrophes, it is permitted to demarcate the string with quotation marks or "double quotes". This is especially useful for SQL statements:

```
$sql = "SELECT `id`, `name` from `people` "
    . "WHERE `name`='Fred' OR `name`='Susan'";
```

This syntax is preferred over escaping apostrophes as it is much easier to read.

Variable Substitution

Variable substitution is permitted using either of these forms:

```
$greeting = "Hello $name, welcome back!";
```

```
$greeting = "Hello {$name}, welcome back!";
```

For consistency, this form is not permitted:

```
$greeting = "Hello ${name}, welcome back!";
```

String Concatenation

Strings must be concatenated using the "." operator. A space must always be added before and after the "." operator to improve readability:

```
$company = 'HUBzero' . ' ' . 'content management system';
```

When concatenating long strings with the "." operator, it is encouraged to break the statement into multiple lines to improve readability. In these cases, each successive line should be padded with white space such that the "."; operator is aligned under the "=" operator:

```
$sql = "SELECT `id`, `name` FROM `users` "
     . "WHERE `name` = 'Jim' "
     . "ORDER BY `name` ASC ";
```

Arrays

Numerically Indexed Arrays

Negative numbers are not permitted as indices.

An indexed array may start with any non-negative number, however all base indices besides 0 are discouraged.

When declaring indexed arrays with the Array function, a trailing space must be added after each comma delimiter to improve readability:

```
$sampleArray = array(1, 2, 3, 'HUBzero');
```

It is permitted to declare multi-line indexed arrays using the "array" construct. In this case, each successive line must be indented to the same level as first line and then padded with spaces such that beginning of each line is aligned:

```
$sampleArray = array(1, 2, 3, 'HUBzero',
                     $a, $b, $c,
                     56.44, $d, 500);
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration:

```
$sampleArray = array(
    1, 2, 3, 'HUBzero',
    $a, $b, $c,
```

```
    56.44, $d, 500,
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

Associative Arrays

When declaring associative arrays with the Array construct, breaking the statement into multiple lines is encouraged. In this case, each successive line must be padded with white space such that both the keys and the values are aligned:

```
$sampleArray = array('firstKey'  => 'firstValue',
                     'secondKey' => 'secondValue');
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration. For readability, the various "=>" assignment operators should be padded such that they align.

```
$sampleArray = array(
    'firstKey'  => 'firstValue',
    'secondKey' => 'secondValue',
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

Classes

- Classes must be named according to HUBzero's naming conventions.
- The brace should always be written on the line underneath the class name.
- Every class must have a documentation block that conforms to the PHPDocumentor

standard.
- All code in a class must be indented with a single tab.
- Only one class is preferred in each PHP file. Additional classes are permitted but strongly discouraged.
- Placing additional code in class files is permitted but discouraged.

The following is an example of an acceptable class declaration:

```
/**
* Documentation Block Here
*/
class SampleClass
{
    // all contents of class
    // must be indented
}
```

Classes that extend other classes or which implement interfaces should declare their dependencies on the same line when possible.

```
class SampleClass extends FooAbstract implements BarInterface
{
}
```

If as a result of such declarations, readability suffers due to line length, break the line before the "extends" and/or "implements" keywords, and pad those lines by one indentation level.

```
class SampleClass
    extends FooAbstract
    implements BarInterface
{
}
```

If the class implements multiple interfaces and the declaration covers multiple lines, break after each comma separating the interfaces, and indent the interface names such that they align.

```
class SampleClass
```

```
    implements BarInterface,
              BazInterface
{
}
```

Class Member Variables

Member variables must be named according to HUBzero's variable naming conventions.

Any variables declared in a class must be listed at the top of the class, above the declaration of any methods.

The var construct is permitted but discouraged. Member variables should declare their visibility by using one of the private, protected, or public modifiers. Giving access to member variables directly by declaring them as public is permitted but discouraged in favor of accessor methods (set & get).

Functions

Declaration

Functions must be named according to HUBzero's function naming conventions.

Methods inside classes must always declare their visibility by using one of the private, protected, or public modifiers.

As with classes, the brace should always be written on the line underneath the function name. Space between the function name and the opening parenthesis for the arguments is not permitted.

Functions in the global scope are strongly discouraged.

The following is an example of an acceptable function declaration in a class:

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
```

```
    public function bar()
    {
        // all contents of function
        // must be indented four spaces
    }
}
```

In cases where the argument list affects readability, you may introduce line breaks. Additional arguments to the function or method must be indented one additional level beyond the function or method declaration. The following is an example of one such situation:

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar($arg1, $arg2, $arg3,
        $arg4, $arg5, $arg6)
    {
        // all contents of function
        // must be indented four spaces
    }
}
```

Note: Pass-by-reference is the only parameter passing mechanism permitted in a method declaration.

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar(&$baz)
    {
```

```
        }
}
```

Call-time pass-by-reference is strictly prohibited.

The return value must not be enclosed in parentheses. This can hinder readability, in additional to breaking code if a method is later changed to return by reference.

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * WRONG
     */
    public function bar()
    {
        return($this->bar);
    }

    /**
     * RIGHT
     */
    public function bar()
    {
        return $this->bar;
    }
}
```

Function and Method Usage

Function arguments should be separated by a single trailing space after the comma delimiter. The following is an example of an acceptable invocation of a function that takes three arguments:

```
threeArguments(1, 2, 3);
```

Call-time pass-by-reference is strictly prohibited. See the function declarations section for the proper way to pass function arguments by-reference.

In passing arrays as arguments to a function, the function call may include the "array" hint and may be split into multiple lines to improve readability. In such cases, the normal guidelines for writing arrays still apply:

```
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'HUBzero',
                     $a, $b, $c,
                     56.44, $d, 500), 2, 3);

threeArguments(array(
    1, 2, 3, 'HUBzero',
    $a, $b, $c,
    56.44, $d, 500
), 2, 3);
```

Control Statements

If/Else/Elseif

Control statements based on the if and else if constructs must have a single space before the opening parenthesis of the conditional.

Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping for larger conditional expressions.

The opening brace is written on the line after the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented using 1 tab.

```
if ($a != 2)
{
    $a = 2;
}
```

If the conditional statement causes the line length to affect readability and has several clauses, you may break the conditional into multiple lines. In such a case, break the line prior to a logic operator, and pad the line such that it aligns under the first character of the conditional clause.

The closing paren in the conditional will then be placed on a line with the opening brace, with one space separating the two, at an indentation level equivalent to the opening control statement.

```
if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d))
{
    $a = $d;
}
```

The intention of this latter declaration format is to prevent issues when adding or removing clauses from the conditional during later revisions.

For if statements that include else if or else, the formatting conventions are similar to the if construct. The following examples demonstrate proper formatting for if statements with else and/or {else if constructs:

```
if ($a != 2)
{
    $a = 2;
}
else
{
    $a = 7;
}

if ($a != 2)
{
    $a = 2;
}
elseif ($a == 3)
{
    $a = 4;
}
else
{
    $a = 7;
}

if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d))
```

```
{
    $a = $d;
}
elseif (($a != $b)
        || ($b != $c))
{
    $a = $c;
}
else
{
    $a = $b;
}
```

PHP allows statements to be written without braces in some circumstances. This is not permitted; all if, else if or else statements must use braces.

Switch

Control statements written with the switch statement must have a single space before the opening parenthesis of the conditional statement and after the closing parenthesis.

All content within the switch statement must be indented one indention level. Content under each case statement must be indented using an additional indention level.

```
switch ($numPeople)
{
    case 1:
        break;

    case 2:
        break;

    default:
        break;
}
```

The construct default should not be omitted from a switch statement.

Note: It is sometimes useful to write a case statement which falls through to the next case by not including a break or return within that case. To distinguish these cases from bugs, any case

statement where break or return are omitted should contain a comment indicating that the break was intentionally omitted.

Inline Documentation

Format

All documentation blocks ("docblocks") must be compatible with the phpDocumentor format. Describing the phpDocumentor format is beyond the scope of this document. For more information, visit: [1]

All class files must contain a "file-level" docblock at the top of each file and a "class-level" docblock immediately above each class.

Files

Every file that contains PHP code must have a docblock at the top of the file that contains these phpDocumentor tags at a minimum:

```
/**
 * @package    hubzero-cms
 * @author     Joe Smith <joesmith@hubzero.org>
 * @copyright  Copyright 2005-2011 Purdue University. All rights rese
rved.
 * @license    http://www.gnu.org/licenses/lgpl-3.0.html LGPLv3
 *
 * Copyright 2005-2011 Purdue University. All rights reserved.
 *
 * This file is part of: The HUBzero(R) Platform for Scientific Collab
oration
 *
 * The HUBzero(R) Platform for Scientific Collaboration (HUBzero) is f
ree
 * software: you can redistribute it and/or modify it under the terms
of
 * the GNU Lesser General Public License as published by the Free Soft
ware
 * Foundation, either version 3 of the License, or (at your option) an
y
 * later version.
 *
 * HUBzero is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
```

```
 *
 * You should have received a copy of the GNU Lesser General Public Li
cense
 * along with this program.  If not, see <http://www.gnu.org/licenses/
>.
 *
 * HUBzero is a registered trademark of Purdue University.
 */
```

Classes

Every class must have a docblock that contains these phpDocumentor tags at a minimum:

```
/**
 * Short description for class
 *
 * Long description for class (if any)...
 *
 * @package     hubzero-cms
 * @subpackage  com_members
 * @copyright   Copyright 2005-2011 Purdue University. All rights rese
rved.
 * @license     http://www.gnu.org/licenses/lgpl-3.0.html LGPLv3
 * @version     Release: @package_version@
 * @since       Class available since Release 1.5.0
 * @deprecated  Class deprecated in Release 2.0.0
 */
```

Functions

Every function, including object methods, must have a docblock that contains at a minimum:

- A description of the function
- All of the arguments
- All of the possible return values

It is not necessary to use the "@access" tag because the access level is already known from the "public", "private", or "protected" modifier used to declare the function.

If a function or method may throw an exception, use @throws for all known exception classes:

```
@throws exceptionclass [description]
```

SQL Queries

SQL keywords are to be written in uppercase, while all other identifiers (which the exception of quoted text) is to be in lowercase.

```
$sql = "SELECT `id`, `name` from `people` "
     . "WHERE `name`='Fred' OR `name`='Susan'";
```

PHP Naming Conventions

Classes

HUBzero Library

HUBzero Core Library uses the [PSR-0](#) class naming convention whereby the names of the classes directly map to the directories in which they are stored. The root level directory ofHUBzero's standard library is the "Hubzero/" directory. All HUBzero core library classes are stored hierarchically under these root directories.

Class names may only contain alphanumeric characters. Numbers are permitted in class names but are discouraged in most cases. Underscores are only permitted in place of the path separator; the filename "Hubzero/User/Profile.php" must map to the class name "HubzeroUserProfile".

If a class name is comprised of more than one word, the first letter of each new word must be capitalized. Successive capitalized letters are not allowed, e.g. a class "HubzeroPDF" is not allowed while "HubzeroPdf" is acceptable.

Note: Code that must be deployed alongside Hubzero and Joomla libraries but is not part of the standard or extras libraries (e.g. application code or libraries that are not distributed by Hubzero) must never start with "Hubzero".

Extensions

Classes should be given descriptive names. Avoid using abbreviations where possible. Class names should always begin with an uppercase letter and be written in CamelCase even if using traditionally uppercase acronyms (such as XML, HTML).

Namespaced

While namespacing an extension is not required, it is encouraged.

Controllers

For components, such as the Blog in the Administrator, the convention is Components[Component]Controllers[Name].

```php
namespace ComponentsBlogControllers;

use HubzeroComponentAdminController;

class Entries extends AdminController
{
    // Methods
```

```
}
```

Models

The naming convention is Components[Component]Models[Name].

```
namespace ComponentsBlogModels;

use HubzeroBaseModel;

class Entry extends Model
{
    // Methods
}
```

Plugins

Currently, plugin naming must follow the pseudo-namespace conventions.

non/pseudo-Namespaced

These conventions define a pseudo-namespace mechanism for extensions in the framework. Third-party developers are to avoid beginning names with 'Hubzero' as it is reserved. It is advisable for developers to name classes with their own unique prefix.

Controllers

For single controller components, the naming convention is [Component]Controller.

```
class ContentController extends HubzeroComponentSiteController
{
    // Methods
}
```

For a multi-controller components, such as the Blog in the Administrator, the convention is [Component]Controller[Name].

```
class BlogControllerEntries extends HubzeroComponentAdminController
```

```
{
    // Methods
}
```

Models

The naming convention is [Component]Model[Name].

```
class BlogModelEntry extends HubzeroBaseModel
{
    // Methods
}
```

Plugins

The naming convention is plg[Folder][Element]

```
class plgContentPagebreak extends HubzeroPluginPlugin
{
    // Methods
}
```

Filenames

Only alphanumeric characters, underscores, and the dash character (”-”) are permitted. Spaces are strictly prohibited.

Any file that contains PHP code should end with the extension ".php". The following examples show acceptable filenames:

```
Hubzero/Session/Helper.php
```

```
Hubzero/View/Helper/Html.php
```

Controllers

For single controller components, the naming convention of

Components[Component][Client]Controller will map to a file name of controller.php and be located in the component folder.

```
com_content
.. /site
.. .. controller.php
```

For a multi-controller components, the convention of Components[Component][Client]Controllers[Name] will map to files located in a /controllers folder under the component folder. The file names will reflect the name of the controller.

```
com_blog
.. /site
.. .. /controllers
.. .. .. entries.php
.. .. .. media.php
```

Models

The naming convention of [Component]Model[Name] will map to a similar file structure. The files will be located in a /models folder under the component folder. The file names will reflect the name of the model.

```
com_blog
.. /models
.. .. entry.php
.. .. comment.php
```

Layouts

Components may support different Layouts to render the data supplied by a View and its Models. A Layout file usually contains markup and some PHP code for display logic only: no functions, no classes.

A Layout consists of at least one .php file and an equally named .xml manifest file located in the /tmpl/ folder of a View, both reflect the internal name of the Layout. The standard Layout is called "display".

```
com_blog
.. /site
.. .. /views
.. .. .. /entries
.. .. .. .. /tmpl
.. .. .. .. .. display.php
.. .. .. .. .. display.xml
.. .. .. .. .. edit.php
.. .. .. .. .. edit.xml
.. .. .. .. .. entry.php
.. .. .. .. .. entry.xml
```

Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted except as a prefix to indicate protected or private methods. Numbers are permitted in function names but are discouraged in most cases.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called "camelCase" formatting.

Verbosity is generally encouraged. Function names should be as verbose as is practical to fully describe their purpose and behavior.

These are examples of acceptable names for functions:

```
filterInput()
```

```
getElementById()
```

```
widgetFactory()
```

```
_myPrivateMethod()
```

For object-oriented programming, accessors for instance or static variables should always be prefixed with "get" or "set". In implementing design patterns, such as the singleton or factory patterns, the name of the method should contain the pattern name where practical to more thoroughly describe behavior.

For methods on objects that are declared with the "private" or "protected" modifier, the first

character of the method name must be an underscore. This is the only acceptable application of an underscore in a method name. Methods declared "public" should never contain an underscore.

Functions in the global scope (a.k.a "floating functions") are permitted but discouraged in most cases. Consider wrapping these functions in a static class.

Variables

Variable names may only contain alphanumeric characters. Underscores and numbers are permitted in variable names but are discouraged in most cases.

For instance variables that are declared with the "private" or "protected" modifier, the first character of the variable name must be a single underscore. Member variables declared "public" should never start with an underscore.

As with function names (see above) variable names must always start with a lowercase letter and follow the "camelCaps" capitalization convention.

Verbosity is generally encouraged. Variables should always be as verbose as practical to describe the data that the developer intends to store in them. Terse variable names such as "$i" and "$n" are discouraged for all but the smallest loop contexts. If a loop contains more than 20 lines of code, the index variables should have more descriptive names.

Names should be descriptive, but concise. We don't want huge sentences as our variable names, but typing an extra couple of characters is always better than wondering what exactly a certain variable is for.

```
namespace HubzeroBase;

class Example
{
    private $_status = null;

    protected $_fieldName = null;

    protected function _sortNames()
    {
        $someNames = array();
    }
}
```

Constants

Constants may contain both alphanumeric characters and underscores. Numbers are permitted in constant names.

All letters used in a constant name must be capitalized, while all words in a constant name must be separated by underscore characters.

For example, EMBED_SUPPRESS_EMBED_EXCEPTION is permitted but EMBED _SUPPRESSEMBEDEXCEPTION is not.

Prefix constant names with the uppercase name of the class/package they are used in. For example, the constants used by the JError class all begin with "JERROR_".

Constants must be defined as class members with the "const" modifier. Defining constants in the global scope with the "define" function is permitted but strongly discouraged.

CSS Coding Styles

Terminology

Concise terminology used in these standards:

```
selector {
    property: value;
}
```

Selectors

Selectors should:

- be on a single line
- end in an opening brace
- be closed with a closing brace on a separate line

A blank line should be placed between each group, section, or block of multiple selectors of logically related styles.

Were appropriate, blocks of related styles should be commented to facilitate understanding of their use.

```
/* Book Navigation */
    .book-navigation .page-next {
    }
    .book-navigation .page-previous {
    }

/* Book Forms */
    .book-admin-form {
        border: 1px solid #000;
    }
```

Note: Indentation is optional but encouraged when commenting blocks of related styles.

Multiple selectors

Multiple selectors should each be on a single line, with no space after each comma:

```
#forum td.posts,
#forum td.topics,
#forum td.replies,
#forum td.pager {
}
```

Properties

All properties should be on the following line after the opening brace. Each property should:

- be on its own line
- be indented one tab relative to the selector line
- have a colon immediately after (no spaces permitted) the property name
- have a single space after the property and before the property value
- end in a semi-colon

```
#forum .description {
    color: #EFEFEF;
    font-size: 0.9em;
    margin: 0.5em;
}
```

Multiple values

Where properties can have multiple values, each value should be separated with a space.

```
font-family: helvetica, sans-serif;
```

Database Schema Conventions

Table Names

Table names have all lowercase letters and underscores between words, also all table names need to be plural, e.g. invoice_items, orders.

If the table name contains serveral words, only the last one should be plural:

```
applications
application_functions
application_function_roles
```

Field Names

Field names will be lowercase, generally singular case, and words are separated by underscores, e.g. order_amount, first_name

Foreign Keys

The foreign key is named with the singular version of the target table name with _id appended to it, e.g. order_id in the items table where we have items linked to the orders table.

Many-To-Many Link Tables

Tables used to join two tables in a many to many relationship is named using the table names they link, with the table names in alphabetical order, for example items_orders.

Indexes

Indexes should follow the naming pattern of idx_{column name}. For example, an index for the column created_by on a table would have an indexed named idx_created_by.

```
ALTER TABLE `#__my_table` ADD INDEX `idx_created_by` (`created_by`);
```

For indexes that use multiple columns, list each column by order of cardinality.

```
ALTER TABLE `#__my_table` ADD INDEX `idx_category_referenceid` (`categ
ory`, `referenced`);
```

Unique Indexes

Unique indexes follow the same pattern as above but should start with uidx_.

```
ALTER TABLE `#__my_table` ADD UNIQUE `uidx_alias` (`alias`);
```

Fulltext Indexes

Fulltext indexes follow the same pattern as above but should start with ftidx_.

```
ALTER TABLE `#__my_table` ADD FULLTEXT `ftidx_content` (`content`);
```

Foundation

Overview

The goal of this document is to give a high-level overview of how the framework works, with more details on some of the major pieces.

Application Structure

The Root Directory

The default application structure of a hub is intended to provide a clean separation of a hub's content, configuration, extensions, and everything else that makes a hub unique from the core framework.

```
/hubzero
.. /administrator
.. /api
.. /app
.. /core
.. muse
.. index.php
.. htaccess.txt
.. robots.txt
```

Admin & API

The administrator and api directories are carry-overs from prior versions of the hub framework and marked for deprecation in a future version fo the framework. Do not place any files or folders within these two directories.

administrator

    The Administrator application, also known as the Back-end, Admin Panel or Control Panel, is the interface where administrators and other site officials with appropriate privileges can manipulate the appearance, enable/disable installed extensions, or manage users and content.

api

    Every hub comes with an API for accessing data from the various components and extensions in a light-weight, speedy manner. This directory contains the entry point to the API and can be accessed by visiting http://{yourhub}.org/api

The App Directory

The brain, or uniqueness, of a hub lives in the app directory. All (non-core) extensions installed, templates, cache files, uploaded content, and configurations will reside in this directory.

When developing extensions for a hub, the [constant](#) PATH_APP should be used for any paths relating to directories or files within the app directory. This is shorter and allows for the potential renaming of the directory while keeping the hub functioning smoothly.

The app directory contains a number of sub-directories used by the hub for managing extensions and files. Most of these directories will initially be empty.

bootstrap
> The bootstrap folder contains a few files that bootstrap the framework and configure available services.

cache
> The cache directory is used for storing generated content. Nothing within is vital but, rather, is used for dramatically improving site performance. The directory is further sub-divided by application type: admin, site, api, cli.

components
> The components directory is where 3rd-party and custom made components will reside.

config
> The config directory, as the name implies, contains all of the hub's configuration files.

logs

modules

plugins

templates

tmp

The Core Directory

If the app directory is the brain, the core directory is the skeleton, muscles, and heart of a hub, containing the framework and numerous pre-installed extensions.

As with the app directory, a global constant of PATH_CORE representing the file path is available.

Constants

System Constants

These constants are defined for use in the CMS and extensions:

DS	Directory separator. "/"
PATH_ROOT	The path to the current installation.
PATH_CORE	The path to the core framework of the CMS.
PATH_APP	The path to the app directory. This is where all a hub's data, custom code, and uploads will reside.

Note: These paths are the absolute paths of these locations within the file system, NOT the path used in a URL.

Service Providers

Overview

Service providers are the central place of application bootstrapping. All of a hub's core services are bootstrapped via service providers.

Every application or "client" type, such as "administrator" or "api", has their own list of services. These are all of the service provider classes that will be loaded for your application. Providers are laze loaded, meaning they will not be loaded on every request, but only when the services they provide are actually needed.

Standard Provider

Service providers must extend the HubzeroBaseServiceProvider class and are required to define at least one method: register(). Aside from the register method, a boot method may also be defined, which allows for a little setup or processing that may need to occur after all services have been registered.

The Register Method

```php
<?php

namespace AppProviders;

use HubzeroBaseServiceProvider;

class FooServiceProvider extends ServiceProvider
{
    /**
     * Register services in the container.
     *
     * @return void
     */
    public function register()
    {
        $this->app['foo'] = function ($app)
        {
            return new Foo();
        });
    }
}
```

The Boot Method

The boot method is called after all other service providers have been registered, giving it access to all other services that have been registered by the framework.

```php
<?php

namespace AppProviders;

use HubzeroBaseServiceProvider;

class FooServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        $this->app['foo']->bar();
    }
}
```

Middleware Provider

A Middleware provider is an extended service provider with a handle method. Rather than extending HubzeroBaseServiceProvider, these providers extend HubzeroBaseMiddleware. While they can register services, they are not required to do so. Instead, they handle (i.e., modify) the incoming request and outgoing response.

The Handle Method

The handle method is called after the application has been booted and accepts a HubzeroHttpRequest object as the only argument.

```php
<?php

namespace AppProviders;

use HubzeroBaseServiceProvider;
```

```
class FooServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function handle(Request $request)
    {
       // Forcefully set the 'foo' var to 'bar'
       $request->setVar('foo', 'bar');

       return $this->next($request);
    }
}
```

Facades

Overview

Facades serve as "static proxies" to underlying classes in the service container. This provides flexibility over traditional static methods with the benefit of terser syntax.

Use

In the context of a hub, a facade is a class that provides access to an object from the container. For this to work, all facades extend the base HubzeroFacadesFacade class.

A facade class only needs to implement a single method: getAccessor, which defines what to resolve from the container. The base Facade class makes use of the __callStatic() magic-method to defer calls from the facade to the resolved object.

Below is the facade for the Filesystem wherein the getAccessor() method returns the string 'filesystem', which is the key that the Filesystem service is registered with on the application.

```
class Filesystem extends Facade
{
    /**
     * Get the registered name.
     *
     * @return  string
     */
    protected static function getAccessor()
    {
        return 'filesystem';
    }
}
```

In the example below, a call is made to Filesystem to check that a file exists. Looking quickly at the code, one might assume that the static method exists() is being called on the Filesystem class:

```
<?php

namespace ComponentsBlogSiteControllers;

use HubzeroComponentSiteController;
```

```
use Filesystem;
use App;

class Media extends SiteController
{
    public function downloadTask()
    {
        //...

        if ( ! Filesystem::exists($file))
        {
            App::abort(404, 'File not found');
        }

        //...
    }
}
```

This facade serves as a proxy to accessing the underlying implementation of the HubzeroFilesystemFilesystem interface. So, when any static method on the facade is referenced, the application resolves the binding from the service container and runs the requested method against that object. In short, any calls made using the facade will be passed to the underlying instance of the filesystem service.

Class Reference

Below is a list of every facade, its underlying class, and the service container binding key where applicable.

Global (all client types)	Facade	Class	Service Key	Client
	App	HubzeroBaseApplicatio n	app	all
	Auth	HubzeroAuthManager	auth	admin, s
	Cache	HubzeroCacheManager	cache	admin, s
	Component	HubzeroComponentLoa der	component	admin, s
	Config	HubzeroConfigReposito ry	config	all
	Date	HubzeroUtilityDate		all
	Document	HubzeroDocumentMan ager	document	admin, s
	Event	HubzeroEventsDispatc	dispatcher	all

Facade	Class	Service Key	Client
Filesystem	HubzeroFilesystemFilesystem	filesystem	all
Html	HubzeroHtmlBuilder	html.builder	admin, s
Lang	HubzeroLanguageTranslator	language	all
Log	HubzeroLogWriter	log.debug	all
Module	HubzeroModuleLoader	module	admin, s
Notify	HubzeroNotificationHandler	notification	admin, s
Pathway	HubzeroPathwayTrail	pathway	site
Plugin	HubzeroPluginLoader	plugin	all
Request	HubzeroHttpRequest	request	all
Response	HubzeroHttpResponse	response	all
Router	HubzeroRoutingRouter	router	all
Session	HubzeroSessionManager	session	admin, s
Toolbar	HubzeroHtmlToolbar	toolbar	admin
Submenu	HubzeroHtmlToolbar	submenu	admin
User	JUser		all

Extensions

Overview

HUBzero CMS is already a rich featured content management system but if you're building a hub and you need extra features which aren't available by default, you can easily extend it with extensions. There are five types of extensions: Components, Modules, Plugins, Templates, and Languages. Each of these extensions handle specific functionality.

Components

The largest and most complex of the extension types, a component is in fact a separate application. A component is a relatively self-contained portion of code with its own functionality, its own database tables and its own presentation. Examples of components are a forum, a blog, a wiki, a photo gallery, etc. One could easily imagine all of these as separate applications or stand-alone systems. A component will be shown in the main part of the website and only one component will be shown. A menu is then in fact nothing more then a switch between different components.

Modules

Modules are extensions which present certain pieces or smaller chunks of information on the site. It is not uncommon to have a number of modules on each web page. A module differs from a component in that it doesn't make sense as a standalone application; Rather, it will just present information or add a functionality to an existing application. Common examples would include displaying the latest blog post on the home page or a search box to be present throughout the site. This is a small piece 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.

Plugins

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 core events or custom events that are triggered from your own code. This is a powerful way of extending the basic functionality.

Templates

A template is a series of files within the Joomla! CMS that control the presentation of the content. The template is not a website; it's also not considered a complete website design. The template is the basic foundation design for viewing your website. To produce the effect of a "complete" website, the template works hand-in-hand with the content stored in the database.

Each hub comes with default templates for both the administrator area and the front-end site.

- **administrator** - kameleon
- **site** - hubbasic2013

Languages

Probably the most basic extensions are languages. Languages can be packaged in two ways, either as a core package or as an extension package. In essence, these files consist key/value pairs, these pairs provide the translation of static text strings which are assigned within the source code. These language packs will affect both the front and administrator side. Note: these language packs also include an XML meta file which describes the language and font information to use for PDF content generation.

Conclusion

If the difference between the three types of extensions is still not completely clear, then it is advisable to go to the admin pages of your installation and check the components menu, the module manager and the plugin manager. A hub comes with a number of core components, modules and plugins. By checking what they're doing, the difference between the three types of building blocks should become clear.

Services

Cache

Overview

A HubzeroCacheManager object is available for managing cache data storage and retrieval.

The Cache service comes with a wrapper class to easily work with multiple cache storage driver instances from a single object.

```
$manager = new HubzeroCacheManager($app);
```

Throughout this documentation the Cache facade will be used as it provides a convenient, terse access to the underlying implementations of the cache manager.

Drivers

A number of cache drivers are available.

Custom Drivers

The extend method on the HubzeroCacheManager can be used to extend the cache facility. It is used to bind a custom driver resolver to the manager. The following example demonstrates how to register a new cache driver named "example":

```
Cache::extend('example', function($config)
{
    return new ExampleStore;
});
```

The first argument passed to the extend method is the name of the driver, which will correspond to the driver option in the config/cache.php configuration file. The second argument is a Closure that should return an HubzeroCacheStorageStorageInterface instance. The closure is passed an array of configuration values.

Retrieving Items

The get method on the Cache facade is used to retrieve items from the cache. If the item does

not exist in the cache, null will be returned.

```
$value = Cache::get('key');
```

A default value can be passed as a second argument to the get method. This value will be returned if the cache store fails to find an item associated with the specified key or the data had expired. The default may be also be a closure:

```
$value = Cache::get('key', 'default');

$value = Cache::get('key', function() { return 'default'; });
```

The all method can be used to retrieve **all** items in the cache store.

```
$data = Cache::all();
```

Checking For Item Existence

The has method may be used to determine if an item exists in the cache:

```
if (Cache::has('key'))
{
    //
}
```

Storing Items

The put method on the Cache object is used to store items in the cache. When placing an item in the cache, the number of minutes for which the value should be cached will also need to specified:

```
Cache::put('key', 'value', $minutes);
```

Alternatively, the add method will only add the item to the cache if it does not already exist in the cache store:

```
Cache::add('key', 'value', $minutes);
```

The forever method may be used to store an item in the cache permanently. These values must be manually removed from the cache using the forget method:

```
Cache::forever('key', 'value');
```

Removing Items

You may remove items from the cache using the forget method on the Cache object:

```
Cache::forget('key');
```

Everything may be removed from the cache store by calling the clean method.

```
Cache::clean();
```

To limit the clean method to specific group of cached data, such as just cached data for the Tags component, a cache group name may be passed. In the example below, this will only remove cached data for the "tags" cache group.

```
Cache::clean('tags');
```

Finally, there is a gc (for "Garbage Collection") method for removing expired data.

```
Cache::gc();
```

Filesystem

Overview

The main entry point for the file system API is the HubzeroFilesystemManager. When working with file systems, this is the class that methods will be invoked on. The Manager makes use of the adapter pattern which helps eliminate the inconsistencies of the different file systems by providing a common interface. So, wether using a Local, Ftp, or Dropbox adapter, the method calls and returned data types will be the same.

Adapters

Local

This is the default file system adapter.

```
$adapter = new HubzeroFilesystemAdapterLocal();

$filesystem = new HubzeroFilesystemFilesystem($adapter);
```

FTP

```
$adapter = new HubzeroFilesystemAdapterFtp(array(
    'host' => 'ftp.example.com',
    'port' => 21,
    'username' => 'username',
    'password' => 'password',
    'root' => '/path/to/root',
));

$filesystem = new HubzeroFilesystemFilesystem($adapter);
```

None

A base adapter used primarily for testing.

```
$adapter = new HubzeroFilesystemAdapterNone();

$filesystem = new HubzeroFilesystemFilesystem($adapter);
```

Retrieving Files

The read method may be used to retrieve the raw string contents of a given file:

```
Filesystem::read('file.jpg');
```

The exists method may be used to determine if a given file exists:

```
if ( ! Filesystem::exists('file.jpg'))
{
    throw new Exception('File not found.');
}
```

Storing Files

The write method may be used to store a file on disk. This method accepts two arguments: the file path and the contents to write.

```
Filesystem::write('file.jpg', $contents);
```

Copy / Move

The copy method may be used to copy an existing file to a new location on the disk:

```
Filesystem::copy('old/file1.jpg', 'new/file1.jpg');
```

The move method may be used to move an existing file to a new location:

```
Filesystem::move('old/file1.jpg', 'new/file1.jpg');
```

Prepend / Append

The prepend method allows for easily prepending contents to the beginning of an existing file.

```
Filesystem::prepend('file.log', 'Prepended Text');
```

Similarly, the append method allows for easily appending contents to the end of a file.

```
Filesystem::append('file.log', 'Appended Text');
```

Removing Files

The delete method accepts a single filename to remove from the system:

```
Filesystem::delete('file.jpg');
```

Directories

soon

Macros

If a feature is not included in the Filesystem class, it can be extended through macros. A macro can extend basic, existing functionality to perform more complex tasks. One example of this would be a macro for creating a directory tree.

Macros must implement HubzeroFilesystemMacroInterface and generally consist of at least two methods: getMethod and handle.

```
class Example extends HubzeroFilesystemMacroBase
{
 public function getMethod()
 {
  return 'examplify';
 }

 public function handle($path)
```

```
 {
  $data = $this->filesystem->read($path);

  return $data . 'EXAMPLE';
 }
}
```

The getMethod method returns the name of the call to the filesystem that will invoke the macro.

The handle method does all the real work. When a macro is invoked, the filesystem object calls handle on the macro and passes it all arguments it received from invoking call.

```
// Add the macro to the filesystem object
$filesystem->addMacro(new Example);

// Append 'EXAMPLE' to a file's contents
$content = $filesystem->examplify('path/to/file');
```

Language

Session

Overview

In its simplest form, a PHP session allows data to be stored temporarily on the server and accessed throughout a user's time on the site. When that user leaves the site or is inactive for a certain amount of time, the data is destroyed.

The Session class has already taken care of many aspects of session storage. It provides a very simple interface to store and retrieve data from the user's session.

Storing Data

Also, when multiple extensions run on a site, there is a possibility of running into naming conflicts in session variables. For this reason, Session allows for the specification of a namespace that a var should be placed under.

```
Session::set('cart', $cart, 'uniqueName');
```

Retrieving Data

Much like in the Request library, a default value can be specified in the get() method as the second argument.

```
// Return an empty array if no data found
$cart = Session::get('cart', array());
```

Events

Overview

Events provide a simple observer implementation, allowing one to subscribe and listen for events in the application.

The Event

When an event is triggered, it's identified by a unique name (e.g. system.onRoute), which any number of listeners might be listening to. An Event instance is also created and passed to all of the listeners.

Naming Conventions

The unique event name can be any string, but optionally follows a few simple naming conventions:

- use only lowercase letters, numbers, dots (.) and underscores (_);
- prefix names with a namespace followed by a dot (e.g. kernel.);
- end names with a verb that indicates what action is being taken (e.g. request).

Event Object

When the dispatcher notifies listeners, it passes an actual Event object to those listeners. The base Event class is very simple: it contains a method for stopping event propagation, methods for adding and removing arguments (i.e., data to be passed to the listeners), and a method for adding to the response.

Often times, data about a specific event needs to be passed along with the Event object so that the listeners have needed information.

```
// Creating an Event called "onSomething".
$event = new HubzeroEventsEvent('system.onDoSomething');

// Adding an argument named "foo" with value "bar".
$event->addArgument('foo', 'bar');
```

Arguments can be added (if not already existing), forcefully set, retrieved, or removed.

addArgument

Add an event argument, only if it is not existing.

setArgument

Set the value of an event argument. If the argument already exists, it will be overridden.

removeArgument

Remove an event argument.

getArgument

Get an event argument value.

hasArgument

Tell if the given event argument exists.

```
class EventListener
{
    public function onDoSomething($event)
    {
        // Check that the event has the necessary 'foo' argument
        if ( ! $event->hasArgument('foo'))
        {
            return;
        }

        // Get the 'foo' argument
        $foo = $event->getArgument('foo');
    }
}
```

The Dispatcher

The dispatcher is the central object of the event dispatcher system. In general, a single dispatcher is created, which maintains a registry of listeners. When an event is triggered via the dispatcher, it notifies all listeners registered with that event:

```
$dispatcher = new HubzeroEventsDispatcher();
```

Registering Listeners

Registering an event listener can be done simply by passing the listener object to the addListener method.

```
$dispatcher->addListener(new SystemListener);
```

By default, the listener will be registered to all events matching its method names. If the listener contains methods that should not be registered, a defined list of events may be passed.

```
$dispatcher->addListener(
    new SystemListener,
    array(
        'onBeforeRoute' => HubzeroEventsPriority::NORMAL,
        'onAfterRoute' => HubzeroEventsPriority::NORMAL,
    )
);
```

In the above example, the SystemListener object is registered as an even listener along with a specified list of events that it listens to. When a defined list of events is passed, a listener's priority for a given Event is also specified.

When a Listener is added without specifying the event names, it is registered with a NORMAL priority to all events. If some listeners have the same priority for a given event, they will be called in the order they were added to the Dispatcher.

It is also possible to register a closure or anonymous function as an event listener:

```
$dispatcher->addListener(
    function($event) {
        $foo = $event->getArgument('foo');
        // ... do cool things here ...
    },
    array(
        'onBeforeRoute' => HubzeroEventsPriority::NORMAL
    )
);
```

Triggering Events

Once listeners and events have been registered, the events can be triggered. The listeners will be called in a queue according to their priority for that Event.

```
// Triggering the onBeforeRoute Event.
$dispatcher->trigger('onBeforeRoute');
```

The trigger method not only accepts an event name but can also accept a custom Event object.

```
// Creating an event called "onBeforeRoute" with a "foo" argument.
$event = new HubzeroEventsEvent('onAfterSomething');
$event->setArgument('foo', 'bar');

$dispatcher->trigger($event);
```

Arguments or data to be passed to the listener can be specified in an array as a second parameter to the trigger.

```
// Triggering the onBeforeRoute Event.
$dispatcher->trigger('onBeforeRoute', array($foo, $bar));
```

In the example above, the two variables $foo and $bar are added as arguments to the event object and passed to the listener. This is functionally equivalent to the following:

```
// Creating an event called "onBeforeRoute" with a "foo" argument.
$event = new HubzeroEventsEvent('onAfterSomething');
$event->setArgument('foo', $foo);
$event->setArgument('bar', $bar);

$dispatcher->trigger($event);
```

Stopping Events

In some cases, it may make sense for a listener to prevent any other listeners from being called. That is, the listener needs to be able to tell the dispatcher to stop all propagation of the event to future listeners. This can be accomplished from inside a listener by calling the stop() method on the event:

```
class SystemListener
{
    public function onBeforeRoute(Event $event)
    {
        // Stopping the Event propagation.
        $event->stop();
    }
```

```
}
```

When stopping the Event propagation, the next listeners in the queue won't be called.

It is possible to detect if an event was stopped by using the isStopped() method which returns a boolean value:

```
class SystemListener
{
    public function onBeforeRoute(Event $event)
    {
        // Stopping the Event propagation.
        $event->stop();

        if ($event->isStopped())
        {
            //...
        }
    }
}
```

The Basics

Requests

Retrieving Input

You may access all input from the global Request instance.

Retrieving Data

If you have a form variable named 'address', you would want to use this code to get it:

```
$address = Request::getVar('address');
```

The DEFAULT Parameter

In the event that 'address' is not in the request or is unset, you may specify a default value as the second argument:

```
$address = Request::getVar('address', 'Address is empty');
echo $address;  // Address is empty
```

The SOURCE Parameter

Frequently, you will expect your variable to be found in a specific portion of the HTTP request (POST, GET, etc...). If this is the case, you should specify which portion; this will slightly increase your extension's security. If you expect 'address' to only be in POST, use this code to enforce that:

```
$address = Request::getVar('address', 'default value goes here', 'post');
```

The VARIABLE TYPE Parameter

The fourth parameter of getVar() can be used to specify certain filters to force validation of specific value types for the variable.

```
$address = Request::getVar('address', 'default value goes here', 'post','variable type');
```

Here is a list of types you can validate:

- INT
- INTEGER
- FLOAT
- DOUBLE
- BOOL
- BOOLEAN
- WORD
- ALNUM
- CMD
- BASE64
- STRING
- ARRAY
- PATH

Cookies

Cookie values may be accessed in the same manner as user input with the one change of method name from input to cookie.

```
$value = Request::cookie('name');
```

Other

The Request class provides many methods for examining the HTTP request for your application.

Request Method

```
$method = Request::method();

if (Request::isMethod('post'))
{
    //
}
```

Current URL

```
$current = Request::current();
```

Root URL for the application

This differs from base() in that it will always return the root URI for the application, regardless of the sub-directory the request was called from.

```
$base = Request::root();
```

Base URL for the application

By default, this method will return the full base path for the current request, including scheme, host, and port. To return the path only, pass true.

```
// Gets http://myhub.org/
$base = Request::base();

// Gets /
$base = Request::base(true);
```

User IP Address

```
$ip = Request::ip();
```

Responses

Overview

The CMS application contains a HubzeroHttpResponse instance that all extension output (component, template, etc) is attached to. The response instance allows for customizing the response's HTTP status code, content, and headers. The response instance inherits from the SymfonyComponentHttpFoundationResponse class, providing a variety of methods for building HTTP responses.

Note: For a full list of available Response methods, check out the [Symfony API documentation](#).

Response Object

The creation, setting of content and headers, and sending of the response is handled automatically by the application. But, in some cases, it is beneficial to access and manipulate the response as needed. The response instance may be accessed via its Facade, available in all application types (site, admin, muse, etc), or by retrieving the object directly from the application container.

Facade:

```
Response::header('Content-Type', 'application/json');

echo json_encode($data);
```

Direct access:

```
$response = App::get('response');
$response->header('Content-Type', 'application/json');

echo json_encode($data);
```

Attaching Headers

Most response methods are chainable, allowing for the fluent building of responses. For example, you may use the header method to add a series of headers to the response before sending it back to the user:

```
$response->header('Content-Type', $type)
            ->header('X-Header-One', 'Header Value')
            ->header('X-Header-Two', 'Header Value');
```

Setting Content

To set the content of the response, use the setContent method.Note that the value passed must be of type string.

```
$response->setContent($output);
```

Sending a Response

This will send the set content and headers to the client.

```
$response->send();
```

Redirects

One may also generate redirects by calling the redirect() method on the App. That method accepts three arguments: 1) a URL to predict to, 2) an optional message to display, and 3) an optional message type.

```
App::redirect(
    Route::url('index.php?option=com_support')
);
```

Note that a redirect call is immediate meaning no code immediately after the redirect will be executed.

```
App::redirect(
    Route::url('index.php?option=com_support')
);

// This will not be executed
```

```
die('Hello');
```

The redirect() method is instantiating a new instance of a a HubzeroHttpRedirectResponse class which is a specialized, extended instance of the HubzeroHttpResponse class. If need be, the class can be directly instantiated:

```
$redirect = new HubzeroHttpRedirectResponse($url);
$redirect->setRequest(App::get('request'));
$redirect->send();
```

Config

Global Configuration

Global (site) configuration values can be directly accessed via the get() method of the global Config facade:

```
$value = Config::get('sitename');
```

Alternatively, one may grab the entire configuration object from the application:

```
$config = App::get('config');
```

```
$value = $config->get('sitename');
```

Component Configuration

Although rarer than accessing the global site configuration, sometimes it is necessary to access component-specific configurations. This can be done through the global Component facade:

```
$config = Component::params('com_mycomponent');
```

Retrieving a value from the configuration:

```
echo $config->get('paramName');
```

Plugin Configuration

A fairly common task is accessing plugin-specific configurations. This can be done by accessing the public params property on all plugins.

```
class plgSystemExample extends Plugin
{
```

```
    public function onDoSomething()
    {
        $config = $this->params;
    }
}
```

If the configuration for a specific plugin is needed from elsewhere (e.g., another extension), this can be done through the global Plugin facade. Call the params() method, passing in the type of plugin (e.g., authentication) and the name (e.g., facebook) of the plugin:

```
$config = Plugin::params('authentication', 'facebook');
```

Retrieving a value from the configuration:

```
echo $config->get('paramName');
```

Module Configuration

Module-specific configurations can be accessed via the public params property on any modules that extend the HubzeroModuleModule class.

```
class Example extends Module
{
    public function display()
    {
        $config = $this->params;
    }
}
```

Retrieving a value from the configuration:

```
echo $config->get('paramName');
```

Languages

Overview

To create your own language file it is necessary that you use the exact contents of the default language file and translate the contents of the define statements. Language files are INI files which are readable by standard text editors and are set up as key/value pairs.

Working With INI Files

INI files have several restrictions. If a value in the ini file contains any non-alphanumeric characters it needs to be enclosed in double-quotes ("). There are also reserved words which must not be used as keys for ini files. These include: NULL, yes, no, TRUE, and FALSE. Values NULL, no and FALSE results in "", yes and TRUE results in 1. Characters {}|&~![()" must not be used anywhere in the key and have a special meaning in the value. Do not use them as it will produce unexpected behavior.

Files are named after their internationally defined standard abbreviation and may include a locale suffix, written as language_REGION. Both the language and region parts are abbreviated to alphabetic, ASCII characters. A user from the USA would expect the language English and the region USA, yielding the locale identifier "en_US". However, a user from the UK may expect a region of UK, yielding "en_UK".

Setup

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

```
; Module - Example (en_US)
MOD_EXAMPLE_HERE_IS_LINE_ONE = "Here is line one"
MOD_EXAMPLE_HERE_IS_LINE_TWO = "Here is line two"
MOD_EXAMPLE_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 {ExtensionPrefix}_{WidgetName}_{TextName} for naming.

Table 1: Translation key prefixes for the various extensions

Extension Type	Key Prefix
Component	

Extension Type	Key Prefix
	Module
	Plugin
	Template

Adhering to this naming convention is not required but is strongly recommended as it can help avoid potential translation collisions. Since a component can potentially have modules loaded into it, the possibility of a widget and a module having the same translation key arises. To illustrate this, we have the following example of a component named mycomponent that loads a module named mymodule.

The language files for both:

```
; mymodule en_US.ini
MYLINE = "Your Line"
```

```
; mycomponent en_US.ini
MYLINE = "My Line"
```

The layout files for both:

```
<!-- mymodule layout -->
<strong><php echo Lang::txt('MYLINE'); ?></strong>
```

```
<!-- mycomponent layout -->
<div>
 <!-- Load the module -->
 <php echo Module::render('mymodule'); ?>
 <!-- Translate some component text -->
 <php echo Lang::txt('MYLINE'); ?>
</div>
```

Outputs:

```
<div>
 <!-- Load the module -->
 <strong>Your Line</strong>
 <!-- Translate some component text -->
 Your Line
</div>
```

Since the module is loaded in the component view, i.e. *after* the component's translation files have been loaded, the module's instance of MYLINE overwrites the existing MYLINE from the component. Thus, the view outputs "Your Line" for the component translation instead of the expected "My Line". Using the HUBzero naming convention of adding component and module name prefixes helps avoid such errors:

The language files for both:

```
; mymodule en-US.ini
MOD_MYMODULE_MYLINE = "Your Line"
```

```
; mycomponent en-US.ini
COM_MYCOMPONENT_MYLINE = "My Line"
```

The view files for both:

```
<!-- mymodule view -->
<strong><php echo Lang::txt('MOD_MYMODULE_MYLINE'); ?></strong>
```

```
<!-- mycomponent view -->
<div>
 <!-- Load the module -->
 <php echo $this->Widgets()->renderWidget('mywidget'); ?>
 <!-- Translate some module text -->
 <php echo Lang::txt('COM_MYCOMPONENT_MYLINE'); ?>
```

```
</div>
```

Outputs:

```
<div>
 <!-- Load the widget -->
 <strong>Your Line</strong>
 <!-- Translate some module text -->
 My Line
</div>
```

To Further avoid potential collisions as it is possible to have a component and module with the same name, module translation keys are prefixed with MOD_ and component translation keys with COM_.

Translating Text

A translate helper (Lang) is available in all views and the appropriate language file for an extension is preloaded when the extension is instantiated. This is all done automatically and requires no extra work on the developer's part to load and parse translations.

Below is an example of accessing the translate helper:

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

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

Overrides

In order to accommodate rewording across hub deployments, we provide a mechanism for overriding language constants. Web developers are highly encouraged to make use of language constants throughout extension development as language overrides are generally simpler and easier to maintain than view overrides when wording simply needs to be updated.

Administrator Interface

1. On the administrative backend, navigate to the Language Manager through the Extensions menu.

2. Click on the Overrides sub-menu.

3. To add a new Language Override, click the "New" button.

To replace a constant, you can search for a value using the right-hand "*Search Text You Want to Change*" functionality. You will need to provide the wording which you wish to replace. The utility will provide you with the Language Constant to be used in the *Create a New Override* interface.

4. Fill out the *Create A New Override* section appropriately.

    a. Unless you have another language installed, leave the Language field to its default setting.

5. The Location field refers to which CMS application this override applies. Possible values are: *Site*, *Administrator*, *Cli*, or *Api*.

6. The File field refers to the location of the language override. The format of this would be <webroot>/<hubname>/app/bootstrap/<application>/language/overrides/<language>-override.ini where values for <application> are: *site, administrator, cli,* or *api.*

File Structure

While the administrative backend offers a user-friendly mechanism to add language overrides, system administrators or developers can directly modify the language file.

Language Overrides are application specific. The HUBzero CMS is comprised of four different application types:

- Site - the public-facing interface of the CMS.

- Administrator - the administrative interface of the CMS.

- Cli - The command line interface (muse is a member of this type).

- Api - The REST-ful-esque API for the HUBzero CMS.

When each application is initiated, a set of parameters are "bootstrapped" to the application. These parameters can be defined in the app/bootstrap directory. One of these parameters is

language.

To change a Language Constant on the public-facing Site:

1. Search for the string which you would like to replace.

    1. Using the command `grep` is useful:

        a. Example: grep -i -H -r -n "one more thing" ./

            i. Returns ./core/components/com_projects/site/language/en-GB/en-GB.com_projects.ini:193:COM_PROJECTS_SETUP_BEFORE_COMPLETE="Just one more thing before you get started... (our lawyers made us do it)"

            ii. The constant is: COM_PROJECTS_SETUP_BEFORE_COMPLETE

2. Edit the file: /www/<webroot>/app/bootstrap/site/language/overrides/en-GB.override.ini using your favorite text editor.

3. Insert the constant and language override into this file.

4. Save the file.

Users & Profiles

User Object

Current User

Accessing the User object for the current user can be done as follows:

```
$user = User::getRoot();
```

Other Users

To access user info for anyone not the current user (accepts user ID number or username):

```
$otheruser = User::getInstance($id);
```

Any field from the user database table may then be accessed through the get('fieldname') method:

```
$id = $user->get('id');
$name = $user->get('name');
```

Object Member Variables and Parameters

These are the relevant member variables automatically generated on a call to getUser():

- **id** - The unique, numerical user id. Use this when referencing the user record in other database tables.
- **name** - The name of the user. (e.g. Jane Doe)
- **username** - The login/screen name of the user. (e.g. janedoe2015)
- **email** - The email address of the user. (e.g. crashoverride@hackers.com)
- **password** - The encrypted version of the user's password
- **password_clear** - Set to the user's password only when it is being changed. Otherwise, remains blank.
- **usertype** - The role of the user within the CMS. (Super Administrator, Editor, etc...)
- **gid** - Set to the user's group id, which corresponds to the usertype.
- **block** - Set to '1' when the user is set to 'blocked'.

- **registerDate** - Set to the date when the user was first registered.
- **lastvisitDate** - Set to the date the user last visited the site.
- **guest** - If the user is not logged in, this variable will be set to '1'. The other variables will be unset or default values.

In addition to the member variables (which are stored in the database in columns), there are parameters for the user that hold preferences. To get one of these parameters, call the getParam() member function of the user object, passing in the name of the parameter you want along with a default value in case it is blank.

```
$language = User::getParam('language', 'the default');

echo "<p>Your language is set to {$language}.</p>";
```

HUBzero Extended Profile

HUBzero comes with extended user profiles that allow for considerably more information than the standard Joomla! User. Extended fields include information about disability, gender, race, bios, picture, etc. To access an extended profile, use the Profile object and load() method (accepts user ID number or username).

```
// Instantiate a new profile object
$profile = new HubzeroUserProfile();

// Load the profile
$profile->load( $id );
```

Alternatively, you may the getInstance() method. This can save on calls to the database as it stores any previously called profiles in memory.

```
// Load the profile
$profile = HubzeroUserProfile::getInstance($id);
```

Any field from the user database table may then be accessed through the get('fieldname') method:

```
$email = $profile->get('email');
$name = $profile->get('name');
```

Multi-option fields such as disability will return arrays.

Checking if a User is logged in

Checking if a user is currently logged in can be done by calling the isGuest() method on the global User facade:

```
// If true, they are logged OUT
// If false, they are logged IN
if (User::isGuest())
{
 return false;
}
```

Alternatively, one may need to work with a user object more directly:

```
// Get the root object behind the facade
$user = User::getRoot();

// ... Do some processing on the $user

if ($user->isGuest())
{
 return false;
}
```

The isGuest() method checks the guest property on the user object. This property can be directly accessed, if desired:

```
// If true, they are logged OUT
// If false, they are logged IN
if (User::get('guest'))
{
 return false;
}
```

Group Memberships

Sometimes you may have a component or plugin that is meant to be accessed by members of a certain group or displays specific data based on membership in certain groups.

```
// Get the groups of the current logged-in user
$user_groups = HubzeroUserHelper::getGroups(User::get('id'));
```

The getGroups() method is passed a user ID and returns an array of objects if any group memberships are found. It will return false if no group memberships are found. Each object contains data specifying the user's status within the group, among other things.

```
Array (
    [0] => stdClass Object (
  [published] => 1
  [cn] => greatgroup
  [description] => A Great Group
  [registered] => 1
  [regconfirmed] => 1
  [manager] => 0
 )
    [1] => stdClass Object (
  [published] => 1
  [cn] => mygroup
  [description] => My Group
  [registered] => 1
  [regconfirmed] => 1
  [manager] => 1
 )
)
```

- **published** - 0 or 1, the published state of the group
- **cn** - string, the group alias
- **description** - string, the group title
- **registered** - 0 or 1, if the user applied for membership to this group (only 0 if the user was invited)
- **regconfirmed** - 0 or 1, if the user's membership application has been accepted (automatically 1 for invitees)
- **manager** - 0 or 1, if the user is a manager of this group

Tags

Overview

The Tag class is a set of tools for adding, removing, editing, and displaying tags on objects. It is used throughout HUB installations for adding tags to such things as resources, users, events, and more.

When properly extended, Tags gives you all of the basic functions you need for managing and retrieving tag records in the database table.

All tags are stored within a single table called "#__tags". The information that associates a particular tag to a specific user, event or group, is stored in a table called "#__tags_object". Storing the association data separate from the tag itself allows for a tag to be represented once but be connected to multiple items. If that tag is ever changed for any reason, it will be represented the same regardless of what object it is attached to.

The #__tags_object table stores, among other things, such data as the unique ID of the tag, the unique ID of the object being tagged, and what component (or, potentially, table) that object belongs to.

id	objectid	tagid	tbl
1	77	6	resources
2	77	6	events

Here we have two entries that both link to a tag with an ID of "6" and both with object IDs of "77". One entry is a resource and the other is an event. The "tbl" field is the most important distinguishing factor; This allows us to have multiple objects with the same object ID, linking to the same tag but not create a conflict.

Writing an extension of Tags

To use Tags, create an extension of the class. In this example, we're adding tags to our "com_example" objects.

```php
<?php
namespace ComponentsExampleModels;

use ComponentsTagsModelsCloud;

require_once(PATH_CORE . DS . 'components' . DS . 'com_tags ' . DS . 'models' . DS . 'cloud.php');

class Tags extends Cloud
```

```
{
  /**
   * Object type, used for linking objects (such as resources) to tags
   *
   * @var string
   */
 protected $_scope = 'example';
}
```

Assign to $_scope the name to be used to uniquely identify tag data as belonging to your specific component.

Using a Tag class extension

Once the class is created and in place, it can be included and instantiated

Create/Update

```
// Retrieve posted tags (comma delimitated string)
$tags = Request::getVar('tags', '');

// Instantiate the tagging class
$cloud = new ComponentsExamplesModelsTags($object_id);

// Tag the object
// $user_id will typically be the current logged in user or User::get(
'id);
$cloud->setTags($tags, $user_id);
```

This method is the same for both adding tags to a previously untagged object and updating the existing list of tags on an object.

Read

render('string')

Returns a string of comma-separated tags.

```
// Instantiate the tagging class
$et = new ComponentsExamplesModelsTags($object_id);

// Get a tag cloud (HTML List)
echo $et->render('string');
```

will give:

```
My Tag, Your Tag, Their Tag
```

render()

Returns a tag cloud, derived of a an HTML list. Each tag is linked to the Tags component and comprises one list item. A CSS class of "tags" on the list allows for styling.

```
// Instantiate the tagging class
$et = new ComponentsExamplesModelsTags($object_id);

// Get a tag cloud (HTML List)
echo $et->render();
```

will give:

```
<ol class="tags">
 <li><a class="tag" href="/tags/mytag">My Tag</a></li>
 <li><a class="tag" href="/tags/yourtag">Your Tag</a></li>
 <li><a class="tag" href="/tags/theirtag">Their Tag</a></li>
</ol>
```

render('array')

Returns an array of associative arrays.

```
// Instantiate the tagging class
$et = new ComponentsExamplesModeslTags($object_id);
```

```
        // Get a tag cloud (HTML List)
        $tags = $et->render('array');
        print_r($tags);
```

will give:

```
Array (
 [0] => Array (
    [tag] => 'mytag'
    [raw_tag] => 'My Tag'
    [tagger_id] => 32
    [admin] => 0
  )
 [1] => Array (
    [tag] => 'yourtag'
    [raw_tag] => 'Your Tag'
    [tagger_id] => 32
    [admin] => 0
  )
 [2] => Array (
    [tag] => 'theirtag'
    [raw_tag] => 'Their Tag'
    [tagger_id] => 32
    [admin] => 0
   )
 )
```

Using the Tag Editor plugin

To make adding tags and editing a list of existing tags in a form, HUBzero offers a Tag Editor plugin. To use the plugin in a view, do the following:

```
// Trigger the event
$tf = Event::trigger( 'hubzer.onGetMultiEntry', array(array('tags','tags','actags','',$tags)) );

// Output
if (count($tf) > 0) {
 echo $tf[0];
```

```
} else {
 echo '<input type="text" name="tags" value="'. $tags .'" />';
}
```

The first parameter passed ('tags') tells the plugin that you wish to display a tags autocompleter. The nest parameter is the name of the input field. The third is the ID of the input field. The fourth is any CSS class you wish to assign to the input. The $tags variable here must be a string of comma-separated tags.

Debugging

Debug Mode

To turn on Debug mode:

- Login to the administration area e.g. http:/YOURSITE/administrator/
- At the top under the **Site** menu click **Global Configuration**.
- Click the **System** tab.
- Under the **Debug Settings** section change **Debug System** to *Yes*.
- Click the **Save** button.

Debug mode will output a list of all queries that were executed in order to generate the page. This will also turn on a stack trace output for error and warning pages. Hubzero components will also have PHP error reporting turned on, allowing one to see any PHP errors that may be present.

Note: Turning on debugging mode for production (live) sites is strongly discouraged and it is recommended to be avoided if at all possible.

Restricting who sees debug output

Since debug mode can contain potentially sensitive, it is **strongly** recommended that access to debug output is restricted to the administrator or super administrator user access levels and/or a defined list of users.

To restrict:

- Login to the administration area e.g. http:/YOURSITE/administrator/
- At the top under the **Extensions** menu click **Plugin Manager**.
- Select **System** from the "Select Type" drop-down.
- Find the debug plugin, typically titled "System - Debug", and click to edit.
- Under the **Parameters** section select the **Allowed Groups** and/or enter a comma-seprated list of usernames into the Allows Users box.
- Click the **Save** button.

Inspecting Variables

Hubzero provides the utility class HubzeroUtilityDebug for dumping variables.

dump()
> This will perform a print_r on the variable passed, wrapping the output in HTML <pre> tags.

ddie()

Short for "dump and die", this will perform a print_r on the variable passed, wrapping the output in HTML <pre> tags and die();.

dlog()

This method allows developers to dump variables to the debug toolbar, allowing data to be inspected without interrupting the flow or process of the code or output. **Note:** This feature requires the global Debug mode and system debug plugin to be enabled.

Example

```
$myvar = array(
    'one' => 'foo',
    'two' => 'bar',
);

HubzeroUtilityDebug::dump($myvar);
```

Illegal variable ... passed to script.

One encounters the following error:

Illegal variable _files or _env or _get or _post or _cookie or _server or _session or globals passed to script.

This error is generated when the key of a key-value pair is numeric in one of the following variables: _files or _env or _get or _post or _cookie or _server or _session or globals. An example of this would be $_POST[5] = 'value'. This is most often generated by having form elements with numeric values as names. For example:

```
<input type="text" name="5" />
```

As the error indicates, this is not allowed. Element names must include at least one non-numeric character. Examples:

```
<input type="text" name="n5" />
```

```
<input type="text" name="n_5" />
```

Scheduled Tasks

Plugins

A set of tasks can be registered with the Cron component by making a plugin. Each plugin must respond to the "onCronEvents" trigger. The response from that trigger is an object (stdClass) that returns the plugin's name and an array of callable tasks (event triggers).

Registering Tasks

Plugins should be placed within the cron plugins folder:

```
/app
.. /plugins
.. .. /cron
```

Here is an example of a cron plugin that registers a set of "mytasks" events.

```
/**
 * Cron plugin for my tasks
 */
class plgCronMytasks extends HubzeroPluginPlugin
{
 /**
  * Return a list of events
  *
  * @return     array
  */
 public function onCronEvents()
 {
  // Load the plugin's language file
  $this->loadLanguage();

  // Create the return object
  $obj = new stdClass();

  // Assign the plugin's name
  $obj->plugin = $this->_name;

  // Build the list of callable events
  $obj->events = array(
```

```
  array(
    'name'   => 'doSomething',  // The name of your task
    'label'  => Lang::txt('PLG_CRON_MYTASKS_DOSOMETHING'),  // Nice la
bel
    'params' => ''  // Name of the params group to load (optional)
   )
  );

  // Return the data
  return $obj;
 }
}
```

As shown in the previous example, each event consist of an array containing three keys: name, label, and params.

name

> The plugin must implement a method with the same name as whatever is specified for the name key and the names should match *exactly*. That is, if a name of 'onJumpUpAndDown' is specified, then the plugin **must** have a method of onJumpUpAndDown();.

label

> This is a nice, human readable name for the event trigger. It should be a language key with an associated string in the plugin's language file.

params

> This is an optional value for specifying a params group (Joomla 1.5) or fieldset (Joomla 1.6+) containing parameters associated with the specific plugin event. This allows for multiple cron jobs calling the same event but with varying values. An example of this can be found in the support tickets cron plugin where the event sendTicketsReminder has a specified params group of 'ticketreminder'. Changing those params would allow, for instance, a job that sends ticket reminders one a month for all open tickets and a ticket reminder once a week for all open and *status: critical* tickets.

> A snippet from the support plugin, specifying the list of available tasks:

```php
/**
 * Cron plugin for support tickets
 */
class plgCronSupport extends HubzeroPluginPlugin
{
 /**
  * Return a list of events
  *
  * @return      array
  */
 public function onCronEvents()
 {
  $this->loadLanguage();

  $obj = new stdClass();
  $obj->plugin = $this->_name;

  $obj->events = array(
   array(
    'name'   => 'onClosePending',
    'label'  => Lang::txt('PLG_CRON_SUPPORT_CLOSE_PENDING'),
    'params' => 'ticketpending'
   ),
   array(
    'name'   => 'sendTicketsReminder',
    'label'  =>  Lang::txt('PLG_CRON_SUPPORT_EMAIL_REMINDER'),
    'params' => 'ticketreminder'
   )
  );

  return $obj;
 }
 ...
}
```

In the support plugin's manifest:

```xml
...
<fieldset group="ticketreminder">
 <field name="support_ticketreminder_severity" type="list" default="all" label="Tickets with severity" description="Ticket severity to message users about.">
  <option value="all">All</option>
```

```
   <option value="critical,major">High</option>
   <option value="normal">Normal</option>
   <option value="minor,trivial">Low</option>
 </field>
 <field name="support_ticketreminder_group" type="text" menu="hid
e" label="For users in group" default="" description="Only users
within the group specified will be messaged." />
</fieldset>
...
```

Running Tasks

All tasks are run as standard plugin events. Tasks should return a boolean of true upon
completion.

See the [managers](#) documentation on how to create and schedule jobs.

Dates

The Date class

To help working with dates the framework provides the Hubzero\Utility\Date class. Since that can be a bit much to type every time when instantiating a new instance, a global Date facade can be used instead. To get a Date object that represents the current date and time do the following:

```
$now = Date::getRoot();
```

The first thing to note is that we do NOT use the =& assignment operator. The static getRoot() method does not return references to globally accessible instances of Date. This means each time Date is used it is retrieving a new object.

It is also possible to specify the date and time we want the Date object to represent. A likely source for this would be a DATETIME field extracted from the database.

```
$created = Date::of($row->created);
```

Since Hubzero\Utility\Date extends PHP's DateTime object, the method used to parse date and time values is relatively robust. Formats other than the MySQL DATETIME representation of YYYY-MM-DD HH:MM:SS can be used. The table below describes the acceptable formats.

Format	Example	Notes
Timestamp	1254497100	Seconds since the Unix Epoch
RFC 2822	Fri, 2 Oct 2009 15:25:00 +0000	Name of day and UTC offset is optional. Date does not support all of the obsolete RFC 822 time zone identifiers. Date support numeric time zone identifiers, UT, GMT, and military time zones.
RFC 3339	2009-10-02 T15:25:00+00:00	RFC 3339 time zone offset can be expressed numerically or as the time zone alpha identifier Z (Zulu, UTC+0). RFC 3339 *is also known as* ISO 8601.
US English date format	2 October 2009	For more information about US

Format	Example	Notes
		English date formats refer to http://php.net/strtotime.

In the table above both the RFC 2822 and RFC 3339 examples include a UTC offset in the value. In the examples the offset is 0. Date always internally represents the date and time in the UTC+0 time zone. Had the offsets in the examples been non zero values, and had we used these to create new Date objects, we would have found that the date and time within the Date objects would have been adjusted to represent a timezone of UTC+0.

Outputting Dates

The Date class includes five handy methods for retrieving formatted date and time strings. The most versatile of these methods is format(). This method allows for explicitly defining the format in which the date and time are to be described. The format can be defined in the same way as when using the PHP function strftime().

```
$string = $myDate->format('%Y-%m-%d');
```

The remaining four methods to retrieve formatted date and time strings are used to extract specific representations of the date and time. These representations are RFC 2822 (successor to RFC 822), ISO 8601 (also known as RFC 3339), Unix timestamp, and SQL (determined by the specific database connector used).

```
// D, d M Y H:i:s
// Tuesday, 06 October 2009 12:54:37+0000
$rfc2822 = $myDate->toRFC822();
```

```
// Y-m-dTH:i:s
// 2009-10-06T12:54:37Z
$iso8601 = $myDate->toISO8601();
```

```
// Unix timestamp
// 1254833677
```

```
$unix = $myDate->toUnix();
```

```
// The format is determined by the databse being used. The following e
xample is for MySQL.
// Y-m-d H:i:s
// 2009-10-06 12:54:37
$mysql = $myDate->toSql();
```

Outputting dates in different time zones

As mentioned above, Date internally stores dates and times in the UTC time zone. In conjunction with that it is good practice to store dates and times in the database in the UTC time zone. For end users however, this is not necessarily easy to read. To aid with this, Date can output dates and times in different time zones.

In addition to the date and time that a Date object represents, a Date object can also record a time zone in which to output formatted dates. This value can be set and retrieved with the setTimezone and getTimezone methods, respectively.

The timezone being discussed in this section is separate from the timezone specified when *creating* a new Date object.

Database

Overview

HUBzero has been built with the ability to use several different kinds of SQL-database-systems and to run in a variety of environments with different table-prefixes. In addition to these functions, the class automatically creates the database connection. Besides instantiating the object, at a minimum, you only need 2 lines of code to get a result from the database in a variety of formats. Using the database layer ensures a maximum of compatibility and flexibility for your extension.

This tutorial looks at how to set and execute various queries.

Configuration

The database configuration for a hub is located at app/config/database.php.

Connections

A hub establishes a database connection, by default, with the configuration specified in app/config/database.php but alternate connections may be established.

```
$mydb = HubzeroDatabaseDriver::getInstance([
    'driver'   => 'pdo',
    'host'     => 'example',
    'user'     => 'example',
    'password' => '******',
    'database' => 'mystuff',
    'prefix'   => 'hub_'
]);
```

Preparing The Query

```
// Get a database object
$db = App::get('db');

$query = "SELECT * FROM `#__example_table` WHERE `id` = 999999;";
$db->setQuery($query);
```

First we instantiate the database object, then we prepare the query. You can use the normal SQL-syntax, the only thing you have to change is the table-prefix. To make this as flexible as possible, Joomla! uses a placeholder for the prefix, the "#__". In the next step, the $db->setQuery(), this string is replaced with the correct prefix.

Now, if we don't want to get information from the database, but insert a row into it, we need one more function. Every string-value in the SQL-syntax should be quoted. For example, MySQL uses back-ticks `` for names and single quotes '' for values. Joomla! has some functions to do this for us and to ensure code compatibility between different databases. We can pass the names to the function $db->nameQuote($name) and the values to the function $db->Quote($value).

A fully quoted query example is:

```
$query = "
  SELECT *
    FROM ".$db->nameQuote('#__example_table')."
    WHERE ".$db->nameQuote('id')." = ".$db->quote('999999').";
  ";
```

Whatever we want to do, we have to set the query with the $db->setQuery() function. Although you could write the query directly as a parameter for $db->setQuery(), it's commonly done by first saving it in a variable, normally $query, and then handing this variable over. This helps writing clean, readable code.

setQuery($query)

The setQuery($query) method sets up a database query for later execution either by the query() method or one of the Load result methods.

```
$db = App::get('db');
$query = "/* some valid sql string */";
$db->setQuery($query);
```

Note: The parameter $query must be a valid SQL string, it can either be added as a string parameter or as a variable; generally a variable is preferred as it leads to more legible code and can help in debugging.

setQuery() also takes three other parameters: $offset, $limit - both used in list pagination; and $prefix - an alternative table prefix. All three of these variables have default values set and can

usually be ignored.

Executing The Query

To execute the query, Joomla! provides several functions, which differ in their return value.

Basic Query Execution

The query() method is the the basic tool for executing sql queries on a database. In the CMS it is most often used for updating or administering the database and not seen often for loading data. This largely because the various load methods detailed on this page have the query step built in to them.

The syntax is very straightforward:

```
$db = App::get('db');
$query = "/* some valid sql string */";
$db->setQuery($query);
$result = $db->query();
```

Note: $db->query() returns an appropriate database resource if successful, or FALSE if not.

Query Execution Information

- getAffectedRows()
- explain()
- insertid()

Insert Query Execution

- insertObject()

Query Results

The database class contains many methods for working with a query's result set.

Single Value Result

loadResult()

Use loadResult() when you expect just a single value back from your database query.

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

This is often the result of a 'count' query to get a number of records:

```
$db = App::get('db');
$query = "
  SELECT COUNT(*)
    FROM ".$db->nameQuote('#__my_table')."
    WHERE ".$db->nameQuote('name')." = ".$db->quote($value).";
  ";
$db->setQuery($query);
$count = $db->loadResult();
```

or where you are just looking for a single field from a single row of the table (or possibly a single field from the first row returned).

```
$db = App::get('db');
$query = "
  SELECT ".$db->nameQuote('field_name')."
    FROM ".$db->nameQuote('#__my_table')."
    WHERE ".$db->nameQuote('some_name')." = ".$db->quote($some_va
lue).";
  ";
$db->setQuery($query);
$result = $db->loadResult();
```

Single Row Results

Each of these results functions will return a single record from the database even though there may be several records that meet the criteria that you have set. To get more records you need

to call the function again.

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

loadRow()

loadRow() returns an indexed array from a single record in the table:

```
...
$db->setQuery($query);
$row = $db->loadRow();
print_r($row);
```

will give:

```
Array (
 [0] => 1
 [1] => John Smith
 [2] => johnsmith@example.com
 [3] => johnsmith
)
```

You can access the individual values by using:

```
$row['index'] // e.g. $row['2']
```

Note:

1. The array indices are numeric starting from zero.
2. Whilst you can repeat the call to get further rows, one of the functions that

returns multiple rows might be more useful

loadAssoc()

loadAssoc() returns an associated array from a single record in the table:

```
$db->setQuery($query);
$row = $db->loadAssoc();
print_r($row);
```

will give:

```
Array (
 [id] => 1
 [name] => John Smith
 [email] => johnsmith@example.com
 [username] => johnsmith
)
```

You can access the individual values by using:

```
$row['name'] // e.g. $row['name']
```

Whilst you can repeat the call to get further rows, one of the functions that returns multiple rows might be more useful

loadObject()

loadObject() returns a PHP object from a single record in the table:

```
$db->setQuery($query);
$result = $db->loadObject();
print_r($result);
```

will give:

```
stdClass Object (
 [id] => 1
 [name] => John Smith
 [email] => johnsmith@example.com
 [username] => johnsmith
)
```

You can access the individual values by using:

```
$row->index // e.g. $row->email
```

Whilst you can repeat the call to get further rows, one of the functions that returns multiple rows might be more useful

Single Column Results

Each of these results functions will return a single column from the database.

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

loadColumn()

loadColumn() returns an indexed array from a single column in the table:

```
$query = "
  SELECT name, email, username
    FROM . . . ";

$db->setQuery($query);
$column= $db->loadColumn();
print_r($column);
```

will give:

```
Array (
 [0] => John Smith
 [1] => Magda Hellman
 [2] => Yvonne de Gaulle
)
```

You can access the individual values by using:

```
$column['index'] // e.g. $column['2']
```

Note:

1. The array indices are numeric starting from zero.
2. loadColumn() is equivalent to loadcolumn(0)

loadColumn($index)

loadColumn($index) returns an indexed array from a single column in the table:

```
$query = "
  SELECT name, email, username
    FROM . . . ";

$db->setQuery($query);
$column= $db->loadColumn(1);
print_r($column);
```

will give:

```
Array (
 [0] => johnsmith@example.com
 [1] => magda_h@example.com
 [2] => ydg@example.com
)
```

You can access the individual values by using:

```
$column['index'] // e.g. $column['2']
```

loadColumn($index) allows you to iterate through a series of columns in the results

```
$db->setQuery($query);
for ( $i = 0; $i <= 2; $i++ ) {
  $column= $db->loadColumn($i);
  print_r($column);
}
```

will give:

```
Array ( [0] => John Smith [1] => Magda Hellman [2] => Yvonne de G
aulle )
Array ( [0] => johnsmith@example.com [1] => magda_h@example.com [
2] => ydg@example.com )
Array ( [0] => johnsmith [1] => magdah [2] => ydegaulle )
```

The array indices are numeric starting from zero.

Multi-Row Results

Each of these results functions will return multiple records from the database.

id	name	email	username

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

loadRowList()

loadRowList() returns an indexed array of indexed arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadRowList();
print_r($row);
```

will give:

```
Array (
 [0] => Array ( [0] => 1 [1] => John Smith [2] => johnsmith@examp
le.com [3] => johnsmith )
 [1] => Array ( [0] => 2 [1] => Magda Hellman [2] => magda_h@exam
ple.com [3] => magdah )
 [2] => Array ( [0] => 3 [1] => Yvonne de Gaulle [2] => ydg@examp
le.com [3] => ydegaulle )
)
```

You can access the individual values by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']['index'] // e.g. $row['2']['3']
```

The array indices are numeric starting from zero.

loadAssocList()

loadAssocList() returns an indexed array of associated arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadAssocList();
print_r($row);
```

will give:

```
Array (
 [0] => Array ( [id] => 1 [name] => John Smith [email] => johnsmi
th@example.com [username] => johnsmith )
 [1] => Array ( [id] => 2 [name] => Magda Hellman [email] => magd
a_h@example.com [username] => magdah )
 [2] => Array ( [id] => 3 [name] => Yvonne de Gaulle [email] => y
dg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']['column_name'] // e.g. $row['2']['email']
```

loadAssocList($key)

loadAssocList($key) returns an associated array - indexed on 'key' - of associated arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadAssocList('username');
print_r($row);
```

will give:

```
Array (
 [johnsmith] => Array ( [id] => 1 [name] => John Smith [email] =>
 johnsmith@example.com [username] => johnsmith )
 [magdah] => Array ( [id] => 2 [name] => Magda Hellman [email] =>
 magda_h@example.com [username] => magdah )
 [ydegaulle] => Array ( [id] => 3 [name] => Yvonne de Gaulle [ema
il] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['key_value'] // e.g. $row['johnsmith']
```

and you can access the individual values by using:

```
$row['key_value']['column_name'] // e.g. $row['johnsmith']['email
']
```

Note: Key must be a valid column name from the table; it does not have to be an Index or a Primary Key. But if it does not have a unique value you may not be able to retrieve results reliably.

loadObjectList()

loadObjectList() returns an indexed array of PHP objects from the table records returned by the query:

```
$db->setQuery($query);
$result = $db->loadObjectList();
print_r($result);
```

will give:

```
Array (
 [0] => stdClass Object ( [id] => 1 [name] => John Smith
      [email] => johnsmith@example.com [username] => johnsmith )
 [1] => stdClass Object ( [id] => 2 [name] => Magda Hellman
      [email] => magda_h@example.com [username] => magdah )
 [2] => stdClass Object ( [id] => 3 [name] => Yvonne de Gaulle
      [email] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']->name // e.g. $row['2']->email
```

loadObjectList('key')

loadObjectList('key') returns an associated array - indexed on 'key' - of objects from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadObjectList('username');
print_r($row);
```

will give:

```
Array (
 [johnsmith] => stdClass Object ( [id] => 1 [name] => John Smith
      [email] => johnsmith@example.com [username] => johnsmith )
 [magdah] => stdClass Object ( [id] => 2 [name] => Magda Hellman
      [email] => magda_h@example.com [username] => magdah )
 [ydegaulle] => stdClass Object ( [id] => 3 [name] => Yvonne de G
aulle
      [email] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['key_value'] // e.g. $row['johnsmith']
```

and you can access the individual values by using:

```
$row['key_value']->column_name // e.g. $row['johnsmith']->email
```

Note: Key must be a valid column name from the table; it does not have to be an Index or a Primary Key. But if it does not have a unique value you may not be able to retrieve results reliably.

Misc Result Set Methods

getNumRows()

getNumRows() will return the number of result rows found by the last query and waiting to be read. To get a result from getNumRows() you have to run it after the query and before you have retrieved any results.

```
$db->setQuery($query);
$db->query();
$num_rows = $db->getNumRows();
```

```
print_r($num_rows);
$result = $db->loadRowList();
```

will give:

3

Note: if you run getNumRows() after loadRowList() - or any other retrieval method - you may get a PHP Warning.

Query Builder

Introduction

HUBzero offers a query builder to help in the abstraction of language-specific SQL syntax, as well as making code more readable. While the query builder itself takes many of its structural queues from SQL, it is not syntax specific. To get started, simply:

```
$query = new HubzeroDatabaseQuery;

// Sample select statement
$users = $query->select('*')
               ->from('#__users')
               ->whereEquals('job', 'programmer')
               ->fetch();
```

Functionally speaking, the query builder serves as a nice intermediary between the feature-rich abstraction that is the ORM, and the execution of raw language specific queries against the database driver. The query builder is actually available on ORM models, and the models will filter all applicable method calls down from the model itself to the underlying query as needed.

Fetching Results

Caching

By default, when fetching results using the query builder, query results will be cached. To disable this behavior, you can pass TRUE as the second argument of the fetch method call.

```
$query->fetch('rows', true);
```

Alternatively, instead of disabling the cache for a single fetch, you can clear the entire cache.

```
Query::purgeCache();
```

Inserting, Updating, and Deleting

When it comes to adding, modifying, or removing records, you have two options for going about this. You can manually build the appropriate query, or you can use one of the shortcut methods. To exemplify this behavior, check out the following two examples.

```
// Full insert
$query = new Query;

$query->insert('users')
      ->values(['name' => 'me', 'email' => 'you@me.com'])
      ->execute();

// Shortcut method
$query->push('users', ['name' => 'me', 'email' => 'you@me.com']);
```

The same principle also applies to updates and deletes:

```
// Full update
$query = new Query;

$query->update('users')
      ->set(['name' => 'you'])
      ->whereEquals('id', 1)
      ->execute();

// Shortcut method
$query->alter('users', 'id', 1, ['name' => 'you']);
```

```
// Full delete
$query = new Query;

$query->delete('users')
      ->whereEquals('id', 1)
      ->execute();

// Shortcut method
$query->remove('users', 'id', 1);
```

Migrations

Overview

HUBzero offers the muse command for automating and simplifying common web developer and system administrator tasks. Of those tasks, running database and content migrations is probably the most crucial to the successful management and deployment of new and updated HUBzero extensions.

The following sections assume that you have the muse command in your path and can execute the script directly. If that is not the case, replace all calls to muse with /www/yourdocroot/muse.

In addition to this documentation, more detailed documentation can always be found by calling: muse migration help

Running Migrations

Running migrations in its basic form is rather simple (though there are a plethora of options available to complicate things if you so desire). Simply type muse migration to run migrations in dry-run mode. This will tell you if you have any pending migrations to run, or if you have perhaps missed a previous migration. If satisfied with what migrations thinks needs to be done, simply run muse migration -f to run the full migration.

That's it!

OK, there's more. By default, migrations won't run migrations that have been missed in the past. To tell migrations to run all pending migrations, irrelevant of date, include the -i option. All other available options can be found by running muse migration help as mentioned above.

Creating Your Own Migrations

This is where the fun begins...

Creating migrations is essential to anyone deploying new extensions in an environment where database tinkering on prod is frowned upon. If the idea of production database access doesn't send chills down your spine, then at least having a migration written will offer a well documented change log for your extensions.

Muse has some basic commands for scaffolding, one of which allows you to create a template migration. To get this auto-generated goodness for yourself, type muse scaffolding create migration -e=extension_name. Here, the extension name would be the extension you are working on, in the form of com_mycomponent or plg_stuff_coolthing. This will drop you into your default editor with the template migration in place and setup according to the HUBzero

conventions of naming and layout.

The migration command will manage what migrations have been run and in what environment. That way you don't have to worry about what you've run and where. That being said, we think it's generally a good idea to make your migrations as foolproof and backwards compatible as possible. To that end, we've added a handful of helper functions to make things as simple as possible. These functions are available on the database object inside of your migration. They are as follows:

```
$this->db->ifTableExists('tableName');
$this->db->ifTableHasField('tableName', 'fieldName');
$this->db->ifTableHasKey('tableName', 'keyName');
```

As an example, instead of just blindly executing an alter table statement to add a new column, you might instead wrap the execution of that statement if an if block that checks for the existence of the table, and the non-existence of the field you want to add...like so:

if ($this->db->tableExists('myTable') && !$this->db->tableHasField('myTable', 'myNewField')) :

Feel free to glance at other migrations in /www/your_doc_root/core/migrations for sample usage.

We've also started adding some additional features to make generating your migrations even easier. So, for example, if you're writing a migration to generate a new table, you can now do muse scaffolding create migration for jos_table_name -e=extension_name. This will create the migration as before, but this time, the migration is completely written for you! We'll add more info here as new features are developed. Also note, the extension name is still required at this time, as the table name and extension name are not explicitly related.

Working with Extensions

Within migrations, there are several helper methods available for common tasks. These methods are also valuable as they abstract out many of the idiosyncrasies of different versions of the database.

These methods primarily include interacting with the extensions tables:

```
$this->addComponentEntry($name);
$this->addPluginEntry($folder, $element);
$this->addModuleEntry($element);
```

```
$this->enablePlugin($folder, $element);
$this->disablePlugin($folder, $element);
```

The counter methods also exist for deleting components, plugins, and modules. These methods are structured the same, simply replace add* with delete*.

Showing Progress

When working with migrations and writing your own, you may occasionally find yourself needing to write a computationally intensive and potentially lengthy migration. When you do this, it's helpful to provide the end user of your migration (maybe other developers or customers) with information on the progress of the migration - rather than making them wonder if something has gone horribly wrong. To accomplish this, you can use the progress callback available on the migration class.

Initialize the progress tracker to show a message about what you're doing.

```
$this->callback('progress', 'init', array('Running ' . __CLASS__ . '.php'));
```

Next, occasionally update the progress notification with your current status.

```
// $i here is a number between 1 and 100 for percentage-based progress
$this->callback('progress', 'setProgress', array($i));
```

Finally, when all is said and done, clean up.

```
$this->callback('progress', 'done');
```

Lastly, instead of a percentage based progress tracker, you can also show a ratio based notification.

```php
// Here we have an example of a ratio with 25 total items (the denominator)
$this->callback('progress', 'init', array('Running ' . __CLASS__ . '.php:', 'ratio', 25));

// Update to 4 (numerator) out of 25 complete
$this->callback('progress', 'setProgress', array(4, 25));

// All done
$this->callback('progress', 'done');
```

ORM

Introduction

Object Relational Mapping, or ORM, is a common paradigm found in many modern CMSs. HUBzero is no exception. HUBzero's ORM is similar to that of many other frameworks, and should be easy to pick up if you've had some experience elsewhere. The goal of HUBzero's ORM is to increase commonality and uniformity in code, while decreasing barriers to entry and errors for developers. While the ORM may not do everything you need it to in some extreme cases, it is relatively full-featured and should greatly speed up the development process.

The Basics

Accessing data

Getting started with the ORM is really quite simple. Your first model could be as basic as an empty class. Consider a User model as our primary working example:

```
use HubzeroDatabaseRelational;

class User extends Relational
{
}
```

With that model in place, you could then loop through all users:

```
foreach (User::all() as $user)
{
    echo $user->name;
}
```

The relational models implement the IteratorAggregate interface, which means that when you start to loop over a model, it will automatically fetch the results for you. If you don't want to loop over the results, you should explicitly tell it to fetch the rows.

```
$users = User::all()->rows();
```

To access a single user, you can load it up by it's primary key. To do so, use one of the "one*" methods.

```
$user = User::one($id);
// or
$user = User::oneOrFail($id);
// or
$user = User::oneOrNew($id);
```

Using the oneOrFail method will result in an exception being thrown if the user ID provided does not exist. The oneOrNew method will result in a blank user being returned if the provided ID does not exist.

In addition to the one*() or all() methods, you also have access to the query builder methods for programmatically limiting the results based on SQL-like constrains. For example:

```
$users = User::whereEquals('name', 'Me');
```

Creating, updating, and deleting

Saving and deleting with the ORM is easy!

```
// Saving/creating a user
$user = oneOrNew(1);
$user->set('name', 'New User');
$user->set('email', 'awesome@gmail.com');
$user->save();

// Deleting a user
$user = oneOrFail(1);
$user->destroy();
```

Don't worry, the relational models will figure out whether you are creating a new model or updating an existing one.

Relationships

Defining relationships between models is a key element to ORMs. The HUBzero ORM offers many standard relationships, including one-one, one-many, and many-many.

One to One

One to one relationships are the simplest variety. To create a one to one relationship between our previous User model and a Phone model, we would add the following method to our class:

```
public function phone()
{
    return $this->oneToOne('Phone');
}
```

Internally, this will attempt to join the Users table to the phone table by way of the user.id and phone.user_id keys. These keys can be overwritten by passing a second or third parameter to the oneToOne method.

It's also important to note that, if you're referencing a model in a different namespace than the current model, you should include the full name-spaced classname.

And to use that relationship, you could do something like this:

```
$user  = User::oneOrFail(1);
$phone = $user->phone;
```

The HUBzero ORM offers dynamic properties. This allows you to simply access the faux property phone, instead of having to explicitly call the phone() method defined above.

One to Many

One to many relationships, though slightly more complex, are probably the most popular relationship scenario. A user may author many posts, or upload many files, or make many comments, and so on. To define a one to many relationship, we:

```
public function posts()
{
    return $this->oneToMany('Post');
}
```

This will join the users table to the posts table by way of users.id and posts.user_id, similar to the way that the one to one relationship works. But, instead of returning a single model, this will return a rows collection of models.

To use this data, you might do something like:

```
$user  = User::oneOrFail(1);
foreach ($user->posts as $post)
{
    echo $post->content;
}
```

Belongs to One

The belongs to one relationship is the inverse of the one to one and one to many relationships. It functions in the same manner as the one to one relationship, but reverses the connection direction (and consequently also reverses the key parameters accepted by the belongsToOne method). Using the above example, a belongs to one relationship between posts and users might look as follows.

```
public function user()
{
    return $this->belongsToOne('User');
}
```

Many to Many

The many to many relationship introduces the relational idea of an an associative entity. Here, we are often trying to mimic a structure such as user roles. Consider the follow table structure.

roles	users	role_user
id	id	id
name	name	role_id
permissions	email	user_id

To accomplish this many to many structure, of users having multiple roles, and a role being assignable to multiple users, you would structure your models like this.

```
// in users model
public function roles()
{
    return $this->manyToMany('Role');
}

// in a roles model
public function users()
{
    return $this->manyToMany('User');
}
```

You'll notice that the many to many model is reversible. You can override the adjoining table name, otherwise it will assume that it is the alphabetically ordered names of your models, separated by an underscore. This is why our table it role_user, not user_role.

One Shifts to Many

In addition to the relationships defined above, HUBzero also offers the ability to handle morphing relationships. This introduces the idea of a conditional join, based on an object id and object scope. Say you have a members table that stores membership for both groups and projects. The table structure might look like this:

groups	projects	members
id	id	id
name	name	scope_id
alias	alias	scope

Our models could then be defined as follows:

```
// in groups model
public function members()
{
    return $this->oneShiftsToMany('Member');
}

// in projects model
public function members()
{
    return $this->oneShiftsToMany('Member');
}
```

Now, instead of joining groups or projects to members by way of projects.id to members.project_id, we associate projects.id to members.scope_id where members.scope is projects.

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
<?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
<?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
<?php
namespace ModulesHelloWorld;

use HubzeroModuleModule;

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
<?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
<?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
<?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
<?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

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 files are copied to the /app/modules/{ModuleName} subfolder of the installation.

Manifest

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
<?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.

Components

Overview

The largest and most complex of the extension types, a component is in fact a separate application. You can think of a component as something that has its own functionality, its own database tables and its own presentation. So if you install a component, you add an application to your website. Examples of components are a forum, a blog, a community system, a photo gallery, etc. You could think of all of these as being a separate application. Everyone of these would make perfect sense as a stand-alone system.

Throughout these articles, we will be using {ComponentName} to represent the name of a component that is variable, meaning the actual component name is chosen by the developer. Notice also that case is important. {componentname} will refer to the lowercase version of {ComponentName}, eg. "CamelCasedController" -> "camelcasedcontroller". Similarly, {ViewName} and {viewname}, {ModelName} and {modelname}, {ControllerName} and {controllername}.

Examples

In the com_drwho example component, we demonstrate working with an MVC structure, basic usage of the database ORM, and more. The admin and site examples show how to output a listing (with pagination), a form for entering new items, and saving to the database.

Other examples included are using multiple controllers, using models, handling errors, adding some security, and pushing assets (e.g., CSS) to the document.

Example usage of the API is also included.

Download: Doctor Who component

Structure

Naming Conventions

The model, view and controller files use classes from the framework, HubzeroBaseModel, HubzeroComponentView and HubzeroComponentSiteController, respectively. Each class is then extended with a new class specific to the component.

Administrative controllers extend HubzeroComponentAdminController which in turn extends HubzeroComponentSiteController and adds a few extra methods frequently used throughout the administrative portion of the site.

All components must be under the Components namespace and follow PSR-0 naming scheme with one exception: files and folders may be lowercase even when their class names are not.

Directories & Files

Components follow the Model-View-Controller (MVC) design pattern. This pattern separates the data gathering (Model), presentation (View) and user interaction (Controller) activities of a module. Such separation allows for expanding or revising properties and methods of one section without requiring additional changes to the other sections.

In its barest state, no database entry or other setup is required to "install" a component. Simply placing the component into the /components directory will make it available for use. However, if a component requires the installation of database tables or configuration (detailed in the config.xml file), then an administrator must install the component using one of the installation options in the administrative back-end.

Note: Components not installed via one of the installation options or without a database entry in the #__extensions table will not appear in the administrative list of available components.

To illustrate the typical component directory structures and files:

```
/app
.. /components
.. .. /com_example
.. .. .. /admin
.. .. .. /api
.. .. .. /helpers
.. .. .. /models
.. .. .. /site
.. .. .. .. /assets
.. .. .. .. .. /css
.. .. .. .. .. /js
```

```
.. .. .. .. .. /img
.. .. .. .. /controllers
.. .. .. .. .. example.php
.. .. .. .. /views
.. .. .. .. .. /example
.. .. .. .. .. .. /tmpl
.. .. .. .. .. .. .. display.php
.. .. .. .. .. .. .. display.xml
.. .. .. .. example.php
.. .. .. .. router.php
```

Files are contained within directories titled "com_example". Some directories and files are optional but, for this example, we've included a more common setup.

All client-specific files and sub-directories are split between the respective client directories, such as admin and site. Since controllers and views are specific to a client, they reside within those client directories. Shared files, typically models and helpers, are within directories at the same level as the client folders.

Directory & File Explanation

/com_{componentname}/{client}/{componentname}.php
 This is the component's entry point for the admin and site clients. API and Cli (console) are special cases and don't require this file.
/com_{componentname}/{client}/views

 This folder holds the different views for the component.

 /com_{componentname}/views/{viewname}
  This folder holds the files for the view {ViewName}.
 /com_{componentname}/views/{viewname}/tmpl

  This folder holds the template files for the view {ViewName}.

  /site/views/{viewname}/tmpl/default.php
   This is the default template for the view {ViewName}.
/com_{componentname}/models

 This folder holds additional models, if needed by the application.

/com_{componentname}/models/{modelname}.php
> This file holds the model class {ComponentName}Model{ModelName}. This class must extend the base class "HubzeroBaseModel". Note that the view named {ViewName} will by default load a model called {ViewName} if it exists. Most models are named after the view they are intended to be used with.

/com_{componentname}/{client}/controllers

This folder holds additional controllers, if needed by the application.

/com_{componentname}/{client}/controllers/{controllername}.php
> This file holds the controller class {ComponentName}Controller{ControllerName}. This class must extend the base class HubzeroComponentsSiteController.

Entry Point

The CMS is always accessed through a single point of entry: index.php for the Site Application or administrator/index.php for the Administrator Application. The application will then load the required component, based on the value of 'option' in the URL or in the POST data. For our component, the URL would be:

```
For search engine friendly URLs:
/hello
```

```
For non-SEF URLs:
/index.php?option=com_hello
```

This will load our main file, which can be seen as the single point of entry for our component: components/com_hello/hello.php.

Implementation

```php
<?php
// Define the namespace
// Components{ComponentName}{ClientName};
namespace ComponentsHelloSite;

// Get the requested controller
$controllerName = Request::getCmd('controller', Request::getCmd('view'
, 'one'));

// Ensure the controller exists
```

```php
if (!file_exists(__DIR__ . DS . 'controllers' . DS . $controllerName .
 '.php'))
{
 App::abort(404, Lang::txt('Controller not found'));
}
require_once(__DIR__ . DS . 'controllers' . DS . $controllerName . '.p
hp');
$controllerName = __NAMESPACE__ . '\Controllers\' . ucfirst(strtolower
($controllerName));

// Instantiate controller
$controller = new $controllerName();
// Execute whatever task(s)
$controller->execute();
```

The first statement is defining the namespace. All component namespaces **must** be under the Components namespace.

__DIR__ is a pre-defined PHP constant that evaluates to the absolute path to the current directory, in our case /webroot/app/components/com_hello/site.

DS is the directory separator of your system: either '/' or ''. This is automatically set by the framework so the developer doesn't have to worry about developing different versions for different server OSs. The DS constant should always be used when referring to files on the local server.

First we look for a requested controller name. There is a default set in case none has been passed or if the requested controller is not found. With the controller name, we build the class name for the controller following the standard namespaced camel-cased pattern of Components{Component name}{Client name}Controllers{Controller name}

After the controller is created, we instruct the controller to execute the task, as defined in the URL: index.php?option=com_hello&task=sometask. If no task is set, the default task 'display' will be assumed. When display is used, the 'view' variable will decide what will be displayed. Other common tasks are save, edit, new...

The main entry point (hello.php) essentially passes control to the controller, which handles performing the task that was specified in the request.

Note that we don't use a closing PHP tag in this file: ?>. The reason for this is that we will not have any unwanted whitespace in the output code. This is default practice and will be used for all php-only files.

Controllers

Overview

The controller is responsible for responding to user actions. In the case of a web application, a user action is (generally) a page request. The controller will determine what request is being made by the user and respond appropriately by triggering the model to manipulate the data appropriately and passing the model into the view. The controller does not display the data in the model, it only triggers methods in the model which modify the data, and then pass the model into the view which displays the data.

Site Controller

```php
<?php
namespace ComponentsHelloSiteControllers;

use HubzeroComponentSiteController;

class One extends SiteController
{
 public function displayTask()
 {
  // Pass the view any data it may need
  $this->view->greeting = 'Hello, World!';

  // Set any errors
  foreach ($this->getErrors() as $error)
  {
   $view->setError($error);
  }

  // Output the HTML
  $this->view->display();
 }
}
```

The first, and most important part to note is that we're extending HubzeroComponentSiteController which brings several tools and some auto-setup for us.

Note: HubzeroComponentSiteController extends HubzeroBaseObject, so all its methods and properties are available.

In the execute() method, the list of available tasks is built from only methods that are 1) public and 2) end in "Task". When calling a task, the "Task" suffix should be left off. For example:

```
// This route
Route::url('index.php?option=com_example&task=other');

// Refers to
....
public function otherTask()
{
 ...
}
....
```

If no task is supplied, the controller will default to a task of "display". The default task can be set in the controller:

```
class One extends SiteController
{
 public function execute()
 {
  // Set the default task
  $this->registerTask('__default', 'mydefault');

  // Set the method to execute for other tasks
  // The following can be called by task=delete and will execute the r
emoveTask method
  $this->registerTask('delete', 'remove');  // (task, method name);

  parent::execute();
 }
 ...
}
```

Each controller extending HubzeroComponentSiteController will have the following properties available:

- _option - String, component name (e.g., com_example)
- _controller - String, controller name
- view - Object (View)

- config - Object (Registry), component config

```php
<?php

class One extends SiteController
{
 public function displayTask()
 {
  $this->view->userName = User::get('name');
  $this->view->display();
 }
}
```

Auto-generation of views

The HubzeroComponentSiteController automatically instantiates a new HubzeroComponentView object for each task and assigns the component ($option) and controller ($controller) names as properties for use in your view. Controller names map to view directory and task names directly map to view names.

```
/{component}
   /site
      /views
         /one  (controller name)
            /tmpl
               /display.php
               /remove.php
```

Example usage within a view:

```php
<p>This is component <?php echo $this->option; ?> using controller: <?php echo $this->controller; ?></p>
```

Changing view layout

As mentioned above, the view object is auto-generated with the same layout as the current $task. There are times, however, when you may want to use a different layout or are executing a task after directing through from a previous task (example: saveTask encountering an error

and falling through to the editTask to display the edit form with error message). The layout can easily be switched with the setLayout method.

```
/{component}
  /views
    /one  (controller name)
      /tmpl
        /display.php
        /world.php
```

```
class One extends SiteController
{
 public function displayTask()
 {
  // Set the layout to 'world.php'
  $this->view->setLayout('world');

  // Output the HTML
  $this->view->display();
 }
}
```

Any assigned data or vars to the view will not be effected.

Admin Controller

Administrator component controls are built and function the same as the Front-end (site) controllers with one key difference: they extends HubzeroComponentAdminController.

```
<?php

class One extends AdminController
{
    ...
}
```

The primary difference between SiteController and AdminController is the pre-defining of a few tasks commonly used in administrator components.

API Controller

API controllers extend HubzeroComponentApiController. Functionally, API controllers are very similar to site and admin controllers in that defining executable tasks is done by creating public methods with a "Task" suffix. They differ, however, in two key ways:

1) Controllers follow a naming convention unique to the API. [TODO: fill in]

2) The API has no concept of views and thus no View object to render data. Instead, data is sent back to the application via the send method which, in turn, prepares the response before delivering to the user.

```php
<?php
namespace ComponentsExampleApiControllers;

use HubzeroComponentApiController;

class Greetings extend ApiController
{
    public function listTask()
    {
        $model = new Archive();
        $data = $model->all();

        $this->send($data);
    }
}
```

Helpers

Overview

A helper class is a class filled with static methods and is usually used to isolate a "useful" algorithm. They are used to assist in providing some functionality, though that functionality isn't the main goal of the application. They're also used to reduce the amount of redundancy in your code.

Implementation

Helper classes are stored in the helpers sub-directory of your component folder. As with all other classes, naming follows the PSR-0 convention and are within the Components namespace. Therefore, our helper class is called ComponentsHelloHelpersOutput.

Here's our com_hello/helpers/output.php helper class:

```php
<?php

namespace ComponentsHelloHelpers;

/**
 * Hello World Component Helper
 */
class Output
{
 /**
  * Method to make all text upper case
  *
  * @param  string  $txt
  * @return  string
  */
 public static function shout($txt='')
 {
  return strToUpper($txt).'!';
 }
}
```

We have one method in this class that takes all strings passed to it and returns them uppercase with an exclamation point attached to the end. To use this helper, we do the following:

```php
<?php

namespace ComponentsHelloSiteControllers;

use HubzeroComponentSiteController;
use ComponentsHelloHelpersOutput;

class Greetings extends SiteController
{
 public function displayTask()
 {
   include_once(dirname(dirname(__DIR__)) . DS . 'helpers' . DS . 'output.php');

   $greeting = Output::shout("Hello World");

   $this->view
                ->set( 'greeting', $greeting )
                ->display();
 }
}
```

Models

Overview

The concept of model gets its name because this class is intended to represent (or 'model') some entity.

Creating A Model

All HUBzero models extend the HubzeroBaseModel class. The naming convention for models in the framework is that the class name starts with the name of the component, followed by 'model', followed by the model name. Therefore, our model class is called ComponentsHelloModelsHello.

```php
<?php
namespace ComponentsHelloModels;

use HubzeroBaseModel;

/**
 * Hello Model
 */
class Hello extends Model
{
    /**
    * Gets the greeting
    *
    * @return string The greeting to be displayed to the user
    */
    public function getGreeting()
    {
        return 'Hello, World!';
    }
}
```

You will notice a lack of include, require, or import calls. Hubzero classes are autoloaded and map to files located in the /core/libraries/Hubzero directory. See more on naming conventions.

Using A Model

Here's an example of using a model with our Hello component (com_hello).

```php
<?php
namespace ComponentsHelloSiteControllers;

use HubzeroComponentSiteController;
use ComponentsHelloModelsHello;

/**
 * Controller for the HelloWorld Component
 */
class Greetings extends SiteController
{
 public function display()
 {
  $model = new Hello();
  $greeting = $model->getGreeting();

  $this->set( 'greeting', $greeting )
       ->display();
 }
}
```

Languages

Setup

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

```
; Module - Hellow World (en-US)
COM_HELLOWORLD_LABEL_USER_COUNT = "User Count"
COM_HELLOWORLD_DESC_USER_COUNT = "The number of users to display"
COM_HELLOWORLD_RANDOM_USERS = "Random Users for Hello World"
COM_HELLOWORLD_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 COM_{ComponentName}_{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("COM_EXAMPLE_MY_LINE"); ?></p>
```

Lang::txt is used for both simple strings and strings that require dynamic data passed to them for variable replacement.

```
<p><?php echo Lang::txt('Hello %s. How are you?', $name); ?></p>
```

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

See the [Languages](#) overview for details.

Views

Directory Structures & Files

Views are written in PHP and HTML and have a .php file extension. View scripts are placed in /com_{component name}/{client}/views/, where they are further categorized by the /{viewname}/tmpl. Within these subdirectories, you will then find and create view scripts that correspond to each controller action exposed; in the default case, we have the view script display.php.

```
/app
  /components
    /com_{componentname}
      /{client [site, admin]}
        /views
          /{viewname}
            /tmpl
              default.php
```

Overriding module and component presentation in templates is further explained in the [Templates: Overrides](#) section.

Creating A View

The task of the view is very simple: It retrieves the data to be displayed and pushes it into the template.

```php
// Instantiate a new view
$view = new HubzeroComponentView(array(
    'name' => $this->_controller,
    'layout' => 'foo'
));

// Assign data to the view
$view->greetings = 'Hello';

// Echo out the results
$view->display();
```

In the above example, the view constructor is passed an array of options. The two most important options are listed: name, which is the folder to look for the view file in and will typically correspond to the current controller's name, and layout, which is the specific view file to load. If no layout is specified, the layout is typically auto-assigned to the current task name. So, if the controller in the example code is one, the directory structure would look as follow:

```
/com_example
    /views
        /one
            /tmpl
                /foo.php
```

Method Chaining

All Hubzero view objects support method chaining for brevity and ease of use.

```php
// Instantiate a new view
$view = new HubzeroComponentView(array(
    'name' => $this->_controller,
    'layout' => 'foo'
));

$view->set('greetings', 'Hello')
    ->setLayout('bar')
    ->display();
```

Assets

Overview

Frequently, components will make use of their styles, images, and scripts to further enhance the interface and user experience. There are a number of helpers to make adding CSS and Javascript to the document a quick and easy process.

Directory Structure & Files

Assets are stored in the same directory as the entry point, views, and controllers for each client type of a component. This means, for example, the administrative side and front-end of a component may make use of completely different assets.

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) to function, eliminating the tedious need for path building and file existence checking before attaching to the document.

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
.. /components
.. .. /{ComponentName}
.. .. .. /{ClientName}
.. .. .. .. /assets
.. .. .. .. .. /css
.. .. .. .. .. /img
.. .. .. .. .. /js
```

Helpers

The HubzeroComponentSiteController and HubzeroComponentView classes bring with them some useful methods for pushing StyleSheets and JavaScript assets to the document and building paths to images. These methods can be called from within a controller or a component 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 component (without the com_ prefix) will be used. For instance, if called within a view of the members component com_members, the system will look for a stylesheet named members.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
<?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
<?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 component'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
.. /components
.. .. /{ComponentName}
.. .. .. /{ClientName}
.. .. .. .. /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" /
>
```

Routing

Overview

All components can be accessed through a query string by using the option parameter which will equate to the name of the component. For example, to access the "Blog" component, you could type http://yourhub.org/index.php?option=com_blog.

When SEF URLs are being employed, the first portion after the site name will almost always be the name of a component. For the URL http://yourhub.org/blog, the first portion after the slash translates to the component com_blog. If a matching component cannot be found, routing will attempt to match against an article section, category, and/or page alias.

While not required, most components will have more detailed routing instructions that allow SEF URLs to be made from and converted back into query strings that pass necessary data to the component. This is done by the inclusion of a file called router.php.

The Router

Every router.php file has a class with two methods: build() which takes a query string and turns it into a SEF URL and parse() which deconstructs a SEF URL back into a query string to be passed to the component.

```php
<?php
namespace ComponentsExampleSite;

use HubzeroComponentRouterBase;

class Router extends Base
{
 public function build(&$query)
 {
  $segments = array();

  if (!empty($query['task']))
  {
   $segments[] = $query['task'];
   unset($query['task']);
  }
  if (!empty($query['id']))
  {
   $segments[] = $query['id'];
   unset($query['id']);
  }
```

```php
  if (!empty($query['format']))
  {
   $segments[] = $query['format'];
   unset($query['format']);
  }

  return $segments;
 }

 public function parse($segments)
 {
  $vars = array();

  if (empty($segments))
  {
   return $vars;
  }
  if (isset($segments[0]))
  {
   $vars['task'] = $segments[0];
  }
  if (isset($segments[1]))
  {
   $vars['id'] = $segments[1];
  }
  if (isset($segments[2]))
  {
   $vars['format'] = $segments[2];
  }

     return $vars;
 }
}
```

The build() Method

This method is called when using Route::url(). Route::url() passes the query string (minus the option={componentname} portion) to the method which returns an array containing the necessary portions of the URL to be constructed *in the order* they need to appear in the final SEF URL.

```php
// $query = 'task=view&id=123&format=rss'
public function build(&$query)
```

```
{
 $segments = array();

 if (!empty($query['task']))
 {
  $segments[] = $query['task'];
  unset($query['task']);
 }
 if (!empty($query['id']))
 {
  $segments[] = $query['id'];
  unset($query['id']);
 }
 if (!empty($query['format']))
 {
  $segments[] = $query['format'];
  unset($query['format']);
 }

 return $segments;
}
```

Will return:

```
Array(
 'view',
 '123',
 'rss'
);
```

This will in turn be passed back to Route::url() which will construct the final SEF URL of example/view/123/rss.

The parse() Method

This method is automatically called on each page view. It is passed an array of segments of the SEF URL that called the page. That is, a URL of example/view/123/rss would be separated by the forward slashes with the first segment automatically being associated with a component name. The rest are stored in an array and passed to parse() which then associates each segment with an appropriate variable name based on the segment's position in the array.

```php
public function parse($segments)
{
 $vars = array();

 if (empty($segments))
 {
  return $vars;
 }
 if (isset($segments[0]))
 {
  $vars['task'] = $segments[0];
 }
 if (isset($segments[1]))
 {
  $vars['id'] = $segments[1];
 }
 if (isset($segments[2]))
 {
  $vars['format'] = $segments[2];
 }

    return $vars;
}
```

Note: Position of segments is very important here. A URL of example/view/123/rss could yield completely different results than a URL of example/rss/view/123.

Configuration

Overview

The framework allows the use of parameters stored in each component.

Defining Options

Configuration options can also be defined in a separate file named config.xml located in the /config sub-directory of the component directory.

```
/app
.. /components
.. .. /com_hello
.. .. .. /config
.. .. .. .. config.xml
```

The XML file's root element should be <config>. Fields are then added and grouped by fieldsets. These fieldsets correspond to the tabs located in the admin side when viewing the component's options.

```xml
<?xml version="1.0" encoding="utf-8"?>
<config>
 <fieldset
  name="greetings"
  label="COM_HELLOWORLD_CONFIG_GREETING_SETTINGS_LABEL"
  description="COM_HELLOWORLD_CONFIG_GREETING_SETTINGS_DESC"
 >
  <field
   name="greeting"
   type="text"
   label="COM_HELLOWORLD_FIELD_GREETING_LABEL"
   description="COM_HELLOWORLD_FIELD_GREETING_DESC"
   default=""
  />
 </fieldset>
</config>
```

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

Retrieving Values

One may quickly retrieve the options for any component by calling the params() method on the Component facade or directly accessing the method on the underlying HubzeroComponentLoader class. This method returns a HubzeroConfigRegistry object.

```
$params = Component::params('com_hello');

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

Packaging

Overview

It is possible to install a component 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 an XML document that will allow the Installer to do this for you. This package file contains a variety of information:

- basic descriptive details about your component (i.e. name), and optionally, a description, copyright and license information.
- a list of files that need to be copied.
- optionally, a PHP file that performs additional install and uninstall operations.
- optionally, an SQL file which contains database queries that should be executed upon install/uninstall

Note: All components must be prefixed with com_.

Manifest

This XML file just lines out basic information about the component 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:

```
<?xml version="1.0" encoding="utf-8"?>
<extension type="component" version="1.5.0">
 <name>hello_world</name>
 <!-- The following elements are optional and free of formatting contt
raints -->
 <creationDate>2007 01 17</creationDate>
 <author>John Doe</author>
 <authorEmail>john.doe@example.org</authorEmail>
 <authorUrl>http://www.example.org</authorUrl>
 <copyright>Copyright Info</copyright>
 <license>License Info</license>
 <!--  The version string is recorded in the components table -->
 <version>Component Version String</version>
 <!-- The description is optional and defaults to the name -->
 <description>Description of the component ...</description>

 <!-- Custom Install Script to execute -->
```

```
 <!-- Note: This will be copied from the root of the installation pack
age to the administrator directory automatically -->
 <installfile>install.eventlist.php</installfile>

 <!-- Custom Uninstall Script to execute -->
 <!-- Note: This will be copied from the root of the installation pack
age to the administrator directory automatically -->
 <uninstallfile>uninstall.eventlist.php</uninstallfile>

 <!-- Install Database Section -->
 <install>
  <sql>
   <file driver="mysql" charset="utf8">install.mysql.utf8.sql</file>
   <file driver="mysql">install.mysql.nonutf8.sql</file>
  </sql>
 </install>

 <!-- Uninstall Database Section -->
 <uninstall>
  <sql>
   <file driver="mysql" charset="utf8">uninstall.mysql.utf8.sql</file>
   <file driver="mysql">uninstall.mysql.nonutf8.sql</file>
  </sql>
 </uninstall>

 <!-- Site Main File Copy Section -->
 <files>
  <filename>index.html</filename>
  <filename>test.php</filename>
  <folder>views</folder>
 </files>

 <!-- Site Main Language File Copy Section -->
 <languages>
  <language tag="en-GB">en-GB.com_test.ini</language>
  <language tag="de-DE">de-DE.com_test.ini</language>
  <language tag="nl-NL">nl-NL.com_test.ini</language>
 </languages>

 <!-- Site Main Media File Copy Section -->
 <media destination="com_test">
  <filename>image.png</filename>
  <filename>flash.swf</filename>
 </media>

 <administration>
```

```
  <!-- Administration Menu Section -->
  <menu img="components/com_test/assets/test-16.png">EventList</menu>
  <submenu>
   <!-- Note that all & must be escaped to & for the file to be valid
XML and be parsed by the installer -->
   <menu link="option=com_helloworld&task=hello&who=world">Hello World
!</menu>
   <!-- Instead of link you can specify individual link attributes -->
   <menu img="icon" task="hello" controller="z" view="a" layout="b" su
b="c">Hello Again!</menu>
   <menu view="test" layout="foo">Testing Foo Layout</menu>
  </submenu>

  <!-- Administration Main File Copy Section -->
  <!-- Note the folder attribute: This attribute describes the folder
    to copy FROM in the package to install therefore files copied
    in this section are copied from /admin/ in the package -->
  <files folder="admin">
   <filename>index.html</filename>
   <filename>admin.test.php</filename>
  </files>

  <!-- Administration Language File Copy Section -->
  <languages folder="admin">
   <language tag="en-GB">en-GB.com_test.ini</language>
   <language tag="de-DE">de-DE.com_test.ini</language>
   <language tag="nl-NL">nl-NL.com_test.ini</language>
  </languages>

  <!-- Administration Main Media File Copy Section -->
  <media folder="admin" destination="com_test">
   <filename>admin-image.png</filename>
   <filename>admin-flash.swf</filename>
  </media>
 </administration>
</extension>
```

Structure

Packaging a component for distribution is relatively easy. The file and directory structure is exactly as it would be after installation. For example, all front-end files are places within a directory called /site and all administration files are placed within a directory called /admin.

Here's what a typical package will look like:

```
/com_{componentname}
  {componentname}.xml
  /site
    {componentname}.php
    controller.php
    /views
      /{viewname}
        /tmpl
          default.php
    /models
      {modelname}.php
    /controllers
      {controllername}.php
  /admin
    {componentname}.php
    controller.php
    /views
      /{viewname}
        /tmpl
          default.php
    /models
      {modelname}.php
    /controllers
      {controllername}.php
```

Just "zip" up the primary directory into a compressed archive file. When the ZIP file is installed, the language file is copied to /app/bootstrap/{client}/language/{LanguageName}/{LanguageName}.{ComponentName}.ini and is loaded each time the module is loaded. All of the other files are copied to the /app/components/{ComponentName} directory of the installation.

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

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 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
<?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  $config  An array that holds the plugin configuration
  * @return  void
  */
 public function __construct(&$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
<?php
// File access check
defined('_HZEXEC_') or die( 'Restricted access' );

Lang::load( 'plg_system_test' );

class plgSystemTest extends HubzeroPluginPlugin
{
 ....
}
```

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:

```php
<p><?php echo Lang::txt("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('_JEXEC') or die('Restricted access'); // no direct acce
ss ?>
<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
<?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
<?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
<?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.

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

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

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

```php
<?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
<?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, $title));
```

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

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

Templates

Overview

A template is a series of files within the CMS that control the presentation of the content. The template is not a website; it's also not considered a complete website design. The template is the basic foundation design for viewing your website. To produce the effect of a "complete" website, the template works hand-in-hand with the content stored in the database.

This article guides you through the process of designing your own template for a HUB. This is intended for web designers/developers with a solid knowledge of CSS and HTML and some basic sense of aesthetics.

Although many currently available HUBs tend to look somewhat similar, you have the freedom to make your HUB look as unique as you want it to be simply by modifying a few CSS and HTML files within your template folder.

Note: All the following articles will refer to construction of a front-end template. However, the concepts, techniques, and methods used also apply to the creation of administrative (back-end) templates unless otherwise noted.

Examples

We have provided an example template that you may use to follow along with the articles or use as a starter for your own HUB template.

Download [Basic Template](#) (zip)

Structure

Overview

All templates should include a manifest in the form of an XML document named templateDetails.xml. The file holds key "metadata" about the template and is essential. Without it, your template won't be seen by the system.

Directory & Files

Templates are found in the /templates directory of a hub's /app. Specific template files are contained within a directory of the same name as the template. While a template may contain any number of files and sub-directories, it must contain at least two files: the primary layout (index.php) and a XML manifest named templateDetails.xml.

```
/app
.. /templates
.. .. /{TemplateName}
.. .. .. /css
.. .. .. /html
.. .. .. /img
.. .. ..  /js
.. .. .. error.php
.. .. .. component.php
.. .. .. index.php
.. .. .. templateDetails.xml
.. .. .. template_thumbnail.png
.. .. .. favicon.ico
```

Designing

Overview

Although many currently available HUBs tend to look somewhat similar, you have the freedom to make your HUB look as unique as you want it to be simply by modifying a few CSS and HTML files within your template folder.

This article makes references to [Adobe Photoshop](#) for creation of design files and images but the developer may use any imaging software they're comfortable with.

Creating A Mock-up

It is recommended to start the design of your HUB template by taking a look at a number of other HUBs and websites and deciding which features are important and best serve the goals of your HUB. Having PIs and other team members involved in the process from the start usually saves much time for defining and polishing the design concept. Once you have a good idea of the look and feel of your HUB and its main features, you would normally create a sketch of the HUB front page in Adobe Photoshop or a similar graphics program. Any secondary page will usually keep the header with the menu and login area, and the footer. For creating the Photoshop mock-up, you are encouraged to use the hubtemplate.psd file attached in the "Examples" section of the Templates Overview. Make sure to get feedback from others and finalize the mock-up before jumping onto the next step.

Page Layout

Overview

A template will typically have two layout files: index.php for the majority of content and error.php for custom error pages ("404 - Not Found", etc.). Both of these files are contained within the top level of a template (i.e., they cannot be placed in a sub-directory of the template).

```
/hubzero
  /templates
    /{TemplateName}
       error.php
       index.php
```

All the HTML that defines the layout of your template is contained in a file named index.php. The index.php file becomes the core of every page that is delivered and, because of this, the file is **required**. Essentially, you make a page (like any HTML page) but place PHP code where the content of your site should go.

The error.php layout, unlike index.php is optional. When not included in a template, Joomla! will use its default system error layout to display site errors such as "404 - Page Not Found". Including error.php is recommended though as it helps give your site a more cohesive feel and experience to the user.

A Breakdown of index.php

Note: For the sake of simplicity, we've excluded some more common portions found in HUBzero templates. The portions removed were purely optional and not necessary for a template to function correctly. We suggest inspecting other templates that may be installed on your HUB for further details.

Starting at the top:

```php
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

$config = JFactory::getConfig();
$juser = JFactory::getUser();

//do we want to include jQuery
```

```
if (JPluginHelper::isEnabled('system', 'jquery'))
{
 $this->addScript($this->baseurl . '/templates/' . $this->template . '
/js/hub.jquery.js');
}
else
{
 $this->addScript($this->baseurl . '/templates/' . $this->template . '
/js/hub.js');
}

// Get the user's browser and browser version
// We add this to the document root as classes for better targeting wi
th CSS
$browser = new HubzeroBrowserDetector();
$b = $browser->name();
$v = $browser->major();

// Set the page title
$this->setTitle($config->getValue('config.sitename') . ' - ' . $this->
getTitle());
?>
<!DOCTYPE html>
<!--[if lt IE 7 ]> <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie6"> <![endif]-->
<!--[if IE 7 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie7"> <![endif]-->
<!--[if IE 8 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie8"> <![endif]-->
<!--[if IE 9 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie9"> <![endif]-->
<!--[if (gt IE 9)|!(IE)]><!--> <html dir="<?php echo $this->direction;
 ?>" lang="<?php echo  $this->language; ?>" class="<?php echo $b . ' '
 . $b . $v; ?>"> <!--<![endif]-->
```

The first line prevents unauthorized people from looking at your coding and potentially causing trouble. Then we grab a reference to the global site configuration. Next, we push some scripts to the document, first checking if the jquery plugin is enabled. Following that, we get the current site visitors browser and browser version. We add this to the document root as classes for better targeting with CSS. The last line of PHP takes the current page title and prepends the site's name. Thus, every page results with a title like "myHUB.org - My Page Title".

The first line of actual HTML tells the browser (and webbots) what sort of page it is. The next

line says what language the site is in.

```
<head>
 <link rel="stylesheet" type="text/css" media="screen" href="<?php
 echo HubzeroDocumentAssets::getSystemStylesheet(array(
   'fontcons', 'reset', 'columns', 'notifications', 'pagination',
   'tabs', 'tags', 'comments', 'voting', 'layout'
 )); /* reset MUST come before all others except fontcons */ ?>" />
 <!-- Include the template's main CSS file -->
 <link rel="stylesheet" type="text/css" media="screen" href="<?php ech
o $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/main.
css" />
 <link rel="stylesheet" type="text/css" media="print" href="<?php echo
 $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/print.
css" />

 <!-- This includes metadata tags and the <title> tag -->
 <jdoc:include type="head" />

 <!--[if IE 9]>
   <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/brow
ser/ie9.css" />
 <![endif]-->
 <!--[if IE 8]>
   <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/brow
ser/ie8.css" />
 <![endif]-->
 <!--[if IE 7]>
   <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/brow
ser/ie7.css" />
 <![endif]-->
</head>
```

The first line compiles several bootstrap CSS files into a single, minified (comments and white-space removed to lessen file size) file to reduce http requests.

The following two lines include the main stylesheet for the template and a print stylesheet that applies more suitable styles when printing.

The fifth line gets Joomla! to put the correct header information in. This includes the page title,

meta information, your main.css, system JavaScript, as well as any CSS or JavaScript that was pushed to the template from an extension (component, module, or plugin). This is a bit different than Joomla! 1.5's typical behavior in that the HUBzero code is automatically finding and including main.css and some key JavaScript files from your template. This is done due to the fact that order of inclusion is important for both CSS and JavaScript. For instance, one cannot execute JavaScript code built using the MooTools framework *before* the framework has been included. It would simply fail. As such, the naming and existence of specific directories, CSS, and JavaScript files becomes quite important for a HUBzero template.

The rest creates links to a couple CSS fix style sheets for Internet Explorer (more on this in the Cascading Style Sheets chapter).

Now for the main body:

```
<body>

 <div id="header">
  <h1><a href="<?php echo $this->baseurl ?>" title="<?php echo $jconfi
g->getValue('config.sitename'); ?>"><?php echo $jconfig->getValue('con
fig.sitename'); ?></a></h1>

  <ul id="toolbar" class="<?php if (!$juser->get('guest')) { echo 'log
gedin'; } else { echo 'loggedout'; } ?>">
<?php
 // Get the current user object
 $juser = JFactory::getUser();

 // Is the user logged in?
 if (!$juser->get('guest')) {
  // Yes. Show them a different toolbar.
  echo '<li id="logout"><a href="/logout"><span>'.JText::_('Logout').'
</span></a></li>';
  echo '<li id="myaccount"><a href="/members/'.$juser->get('id').'"><s
pan>'.JText::_('My Account').'</span></a></li>';
  echo '<li id="usersname">'.$juser->get('name').' ('.$juser->get('use
rname').')</li>';
 } else {
  // No. Show them the login and register options.
  echo "ttt".'<li id="login"><a href="/login" title="'.JText::_('Login
').'">'.JText::_('Login').'</a></li>'."n";
  echo "ttt".'<li id="register"><a href="/register" title="'.JText::_(
'Sign up for a free account').'">'.JText::_('Register').'</a></li>'."n
";
 }
?>
  </ul>
```

```
  <!-- Include any modules for the "search" position -->
  <jdoc:include type="modules" name="search" />
 </div><!-- / #header -->

 <!-- Include any modules assigned to the "user3" position -->
 <div id="nav">
  <h2>Navigation</h2>
  <jdoc:include type="modules" name="user3" />
 </div><!-- / #nav -->

 <div id="wrap">
  <div id="content" class="<?php echo $option; ?>">
   <!-- Include the component output -->
   <jdoc:include type="component" />
  </div><!-- / #content -->

  <div id="footer">
   <!-- Include any modules assigned to the "footer" position -->
   <jdoc:include type="modules" name="footer" />
  </div><!-- / #footer -->
 </div><!-- / #wrap -->
</body>
```

First we layout the site's masthead in the <div id="header"> block. Inside, we set the <h1> tag to the site's name, taken from the global site configuration.

Next, we move on to a toolbar that is present in the masthead of every page. This toolbar contains "login" and "register" links when not logged in and "logout" and "My Account" links when logged in. While not required, it is highly recommended that all templates include some form of this arrangement in an easy-to-find, consistent location.

Some modules that have been assigned the position "search" are then loaded in the masthead. Most HUBzero templates default to having a simple search form module appear. Again, this is not required and placement of modules is entirely up to the developer(s) but we, once again, strongly recommend that some form of a search box be included on all pages.

Then we move on to a block where navigation is loaded. It is here that our main menu will appear.

Next, we get to the primary content block. One of the first things you may notice is the use of module as a jdoc:include type. This is how we tell where in our template to output modules that have been assigned to specific positions.

It is also worth noting the small bit of PHP (<?php echo $option; ?>) in the class attribute of the content <div>. This small bit of code outputs the name of the current component as a CSS class. So, if one were on a page of a "groups" component, the resulting HTML would be <div id="content" class="com_groups">. Since all component output is contained inside the "content" div, this allows for more specific CSS targeting.

See the [Modules: Loading](#) article for more details on module positioning.

The content div contains a very important jdoc:include of type component. This is where all component output will be injected in the template. It is essential this line be included in a template for it to be able to display any content.

A Breakdown of error.php

Starting at the top:

```php
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

// Get the user's browser and browser version
// We add this to the document root as classes for better targeting wi
th CSS
$browser = new HubzeroBrowserDetector();
$b = $browser->name();
$v = $browser->major();

// Get the site config
$jconfig = JFactory::getConfig();
?>
<!DOCTYPE html>
<!--[if lt IE 7 ]> <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie6"> <![endif]-->
<!--[if IE 7 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie7"> <![endif]-->
<!--[if IE 8 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie8"> <![endif]-->
<!--[if IE 9 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie9"> <![endif]-->
<!--[if (gt IE 9)|!(IE)]><!--> <html dir="<?php echo $this->direction;
 ?>" lang="<?php echo  $this->language; ?>" class="<?php echo $b . ' '
 . $b . $v; ?>"> <!--<![endif]-->
```

The first line prevents unauthorized people from looking at your coding and potentially causing trouble. Then we grab a reference to the global site configuration. The first line of actual HTML tells the browser (and webbots) what sort of page it is. The next line says what language the site is in.

```
<head>
 <meta http-equiv="content-type" content="text/html; charset=utf-8" />
 <title><?php echo $jconfig->getValue('config.sitename'); ?> - <?php e
cho $this->title; ?> - <?php echo $this->error->message ?></title>
 <link rel="stylesheet" type="text/css" media="all" href="<?php echo $
this->baseurl ?>/templates/<?php echo $this->template; ?>/css/error.cs
s" />
</head>
```

Unlike with index.php, we do not include the <jdoc:include type="head" /> tag. Instead, we simply set a single metadata tag to declare the character set and then set the title tag. Next, we include the error.css style sheet, which contains styling just for this layout.

Now for the main body:

```
<body>
 <div id="wrap">
  <div id="header">
   <h1><a href="<?php echo $this->baseurl ?>" title="<?php echo $confi
g->getValue('config.sitename'); ?>"><?php echo $config->getValue('conf
ig.sitename'); ?></a></h1>
  </div>
  <div id="outline">
   <div id="errorbox" class="code-<?php echo $this->error->code ?>">
    <h2><?php echo $this->error->code ?> - <?php echo $this->error->me
ssage ?></h2>

    <p><?php echo JText::_('You may not be able to visit this page bec
ause of:'); ?></p>

    <ol>
     <li><?php echo JText::_('An out-of-
date bookmark/favourite'); ?></li>
     <li><?php echo JText::_('A search engine that has an out-of-
date listing for this site'); ?></li>
     <li><?php echo JText::_('A mis-typed address'); ?></li>
     <li><?php echo JText::_('You have no access to this page'); ?></l
i>
```

```
    <li><?php echo JText::_('The requested resource was not found');
?></li>
    <li><?php echo JText::_('An error has occurred while processing y
our request.'); ?></li>
  </ol>

  <p><?php echo JText::_('If difficulties persist, please contact th
e system administrator of this site.'); ?></p>
  </div><!-- / #errorbox -->

  <form method="get" action="/search">
   <fieldset>
    <?php echo JText::_('Please try the'); ?> <a href="index.php" tit
le="<?php echo JText::_('Go to the home page'); ?>"><?php echo JText::
_('Home Page'); ?></a> <span><?php echo JText::_('or'); ?></span>
    <label>
      <?php echo JText::_('Search:'); ?>
      <input type="text" name="searchword" value="" />
    </label>
    <input type="submit" value="<?php echo JText::_('Go'); ?>" />
   </fieldset>
  </form>
  </div><!-- / #outline -->
<?php
  if ($this->debug) :
   echo "tt".'<div id="techinfo">'."n";
   echo $this->renderBacktrace()."n";
   echo "tt".'</div>'."n";
  endif;
?>
 </div><!-- / #wrap -->
</body>
```

As can be seen, this is relatively straight-forward. We set a title for the page, output the error message, provide some potential reasons for the error and, finally, include a search form. Note that we did not use any modules.

One portion to pay special attention to is the small bit of PHP at the end of the page. This outputs a stack trace when site debugging is turned on.

Note: It is never recommended to turn on debugging on a production site.

Loading Modules

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" />
```

See the [Modules: Loading](#) article for further details on how to use more advanced features.

Cascading Style Sheets

Overview

CSS stands for Cascading Style Sheet. HTML tags specify the graphical flow of the elements, be it text, images or flash animations, on a webpage. CSS allows us to define the appearances of those HTML tags with their content, somewhere, so that other pages, if want be, may adhere to. This brings along consistency throughout a website. The cascading effect stipulates that the style of a tag (parent) may be inherited by other tags (children) inside it.

Professional websites separate styling from content. There are many reasons for this, the most obvious (to a developer) being the ability to control the appearance of many pages by changing one file. Styling information includes: fonts, backgrounds, images (that recur on every page), position and dimensions of elements on the page. Your HTML file will now be left with: header information; a series of elements; the text of your website. Because you are creating a Joomla! template, you will actually have: some header information, PHP code to request the rest of the header information, a series of elements, PHP code to request each module position, and PHP code to request the main content.

Style information is coded in CSS and usually stored in files with the suffix .css. A webpage contains a link to the associated .css file so a browser can find the appropriate style information to apply to the page. CSS can also be placed inside a HTML file between <style type="text/css"></style> tags. This is, however, discouraged as it is mixing style and content elements which can make future changes more difficult.

Implementation

Definitions for this section:

External CSS files
      using <link> in the <head>
Document head CSS
      using <style> in the <head>
Inline CSS
      using the style attribute on a tag, i.e. <div style="color:red;">

Guidelines

1. External CSS files should be used in preference to document head CSS and document head CSS should be used in preference to inline CSS.
2. CSS files MUST have the file extension .css and should be stored in the relevant includes directory in the site structure, usually /style/.
3. The file size of CSS files should be kept as low as possible, especially on high demand

pages.

4. External CSS must be linked to using the <link> element which must be placed in the head section of the document. This is the preferred method of using CSS. It offers the best experience for the user as it helps prevent FOUC (flash of unstyled content), promotes code reuse across a site and is cacheable.

5. External style sheets should not be imported (i.e. using @import) as it impairs caching. In IE @import behaves the same as using <link> at the bottom of the page (preventing progressive rendering), so it's best not to use it. Mixing <link> and @import has a negative effect on browsers' ability to asynchronously download the files.

6. Document head CSS may be used where a style rule is only required for a specific page.

7. Inline styles should not be used.

8. Query string data (e.g. "style.css?v=0.1") should not be used on an external CSS file. Use of query strings on CSS files prevents them from caching in some browsers. Whilst this may be desirable for testing, and of course may be used for that, it is very undesirable for production sites.

Directory & Files

Convention places CSS files within a directory named css inside the template directory. While developers are not restricted to this convention, we do recommend it as it helps keep the layout and structure of HUBzero templates consistent. A developer from one project will instantly know where to find certain files and be familiar with the directory structure when working on a project originally developed by someone else.

There are a handful of common CSS files found among most HUBzero. While none of these are required, it is encouraged to follow the convention of including them as it promotes consistency among HUBzero templates and comes with the advantage that certain files, such as main.css are auto-loaded, thus reducing some work on the developer's part.

Here's the standard directory and files for CSS found in a HUBzero template:

```
/hubzero
  /templates
    /{TemplateName}
      /css
        error.css
        browser/ie7.css
        browser/ie8.css
        browser/ie9.css
        main.css
        print.css
        component.css
```

File details:

error.css
>   This is the primary stylesheet loaded by error.php.
ie8.css
>   Style fixes for Internet Explorer 8.
ie7.css
>   Style fixes for Internet Explorer 7.
ie9.css
>   Style fixes for Internet Explorer 9.
main.css
>   This is the primary stylesheet loaded by index.php. The majority of your styles will be in here.
print.css
>   Styles used when printing a page.
component.css
>   This file is meant to be included **before** any other CSS file. Its purpose is to reduce browser inconsistencies in things like default line heights, margins and font sizes of headings, and so on.

Bootstrap

Several bootstrap styles are available in the core, broken into individual stylesheets to make it easier for you to decide what styles you do and do not want to incorporate into your template.

The bootstrap stylesheets can be found in the /media/system/css directory and can be linked to or imported like any other stylesheet. However, for sake of site performance, we recommend using the HubzeroDocumentAssets::getSystemStylesheet() method. This method accepts wither a comma-separated string or array of core stylesheets to include and then compiles them into a single file with comments and white-space stripped out. The resulting file is saved in the cache with a timestamp. Should any of the core files change, the resulting compiled stylesheet will automatically be updated. This has two immediate advantages of 1) fewer http requests (improves page load time) and 2) ensures browsers re-cache the CSS whenever it has changed.

Example usage:

```
<link rel="stylesheet" type="text/css" media="screen" href="<?php echo
 HubzeroDocumentAssets::getSystemStylesheet(array(
 'reset',
 'fontcons',
 'columns',
 'notifications',
```

```
  'pagination',
  'tabs',
  'tags',
  'comments',
  'voting',
  'layout'
)); ?>" />
```

reset.css

> This file is meant to be included **before** any other CSS file. Its purpose is to reduce browser inconsistencies in things like default line heights, margins and font sizes of headings, and so on.
>
> The reset styles given here are intentionally very generic. There isn't any default color or background set for the <body> element, for example. Colors and any other styling should be addressed in the template's primary stylesheet after loading reset.css.

fontcons.css

> This is a custom created icon (dingbat) font used for many of the icons found throughout a hub.

columns.css

> This sets up basic structure for generating layouts that use columns. It supports up to twelve columns and any combination there in. See usage.

notifications.css

> Default styles for warning, error, help, and info messages.

pagination.css

> Basic styling for pagination.

tabs.css

> Default styles for a menu (list) displayed as tabs.

tags.css

Tag styles. Tags are used frequently throughout a hub and this stylesheet helps ensure the look consistent.

comments.css

Comments appear on many items such as KB articles, Questions and Answers, Support tickets, Forums, Blog posts, and more. This is a stylesheet for handling basic layout and styles of a list of (nested) comments and the form for submitting comments.

voting.css

Basic styles for thumbs-up and thumbs-down voting buttons.

layout.css

Default styles for containers, result lists, and other basic structural items used frequently in a hub.

Typical main.css Structure

main.css controls base styling for your HUB, which is usually further extended by individual component CSS.

We took every effort to organize the main.css in a manner allowing you to easily find a section and a class name to modify. E.g. if you want to change the way headers are displayed, look for "headers" section as indicated by CSS comments. Although you can modify all existing classes, depending on your objectives, it is recommended to avoid modifications to certain sections, as indicated below. While you can add new classes as needed, we caution strongly about removing or renaming any of the existing IDs and classes. Many HUBzero components take advantage of these code styles and any alterations made risk breaking the template display.

Some sections that you are likely to modify:

```
Body - may want to change site background or font family.
Links - pick colors for hyperlinks
Headers - pick colors and font size of headings
```

```
Lists - may want to change general list style
Header - you will definitely want to change this
Toolbar - display of username, login/logout links etc.
Navigation - display of main menu
Breadcrumbs - navigation under menu on secondary pages
Extra nav - links that appear on the right-
hand side in multiple components
Footer
```

Sections where you would want to avoid serious modifications:

```
Core classes
Site notices, warnings, errors
Primary Content Columns
Flexible Content Columns
Sub menu - display of tabs in multiple components
```

print.css

This is a style sheet that is used only for printing. It removes unnecessary elements such as menus and search boxes, adjusts any background and font colors as needed to improve readability, and can expose link URLs through generated content (advanced browsers only, e.g. Safari, Firefox).

error.css

This is a style sheet that is used only by the error.php layout. It allows for a more custom styling to error pages such as "404 - Page Not Found".

Internet Explorer

We strongly encourage developers to test their templates in as many browsers and on as many operating systems as possible. Most modern browsers will have little differences in rendering,

however, Internet Explorer deserves special mention here.

The most widely used browser, Internet Explorer, is also one of the most lacking in terms of CSS support. Internet Explorer has also, traditionally, handled rendering of block elements, element positioning, and other common tasks a bit differently than many other browsers. As can be expected, this has led to much controversy and discussion on how best to handle such differences. We strongly recommend designing for and testing your templates in alternate browsers such as [Safari](), [Firefox](), [Chrome](), or [Opera]() and then applying fixes to Internet Explorer afterwards. We recommend the use of conditional comments to apply special Internet Explorer only stylesheets.

..1a Conditional Comments

Conditional comments only work in Internet Explorer on Windows, and are thus excellently suited to give special instructions meant only for Internet Explorer on Windows. They are supported from Internet Explorer 5 onwards, and it is even possible to distinguish between versions of the browser.

Conditional comments work as follows:

```
<!--[if IE 6]>
 Special instructions for IE 6 here
<![endif]-->
```

Their basic structure is the same as an HTML comment (<!-- -->). Therefore all other browsers will see them as normal comments and will ignore them entirely. Internet Explorer, however, recognizes the special syntax and parses the content of the conditional comment as if it were normal page content. As such, they can contain any web content you wish to display only to Internet Explorer. While we're using this feature to load CSS files, it can also be used to load JavaScript or display Internet Explorer specific HTML.

Note: Since conditional comments use the HTML comment structure, they can only be included in HTML, and not in CSS files.

Conditional comments support some variation in syntax. For example, it is possible to target a specific browser version as demonstrated above or target multiple versions such as "all versions of Internet Explorer lower than 7". This can be done with a couple handy operators:

- gt = greater than
- gte = greater than or equal to
- lt = less than
- lte = less than or equal to

```
<!--[if IE]>
 According to the conditional comment this is Internet Explorer
<![endif]-->
<!--[if IE 5]>
 According to the conditional comment this is Internet Explorer 5
<![endif]-->
<!--[if IE 5.0]>
 According to the conditional comment this is Internet Explorer 5.0
<![endif]-->
<!--[if IE 5.5]>
 According to the conditional comment this is Internet Explorer 5.5
<![endif]-->
<!--[if IE 6]>
 According to the conditional comment this is Internet Explorer 6
<![endif]-->
<!--[if IE 7]>
 According to the conditional comment this is Internet Explorer 7
<![endif]-->
<!--[if IE 8]>
 According to the conditional comment this is Internet Explorer 8
<![endif]-->
<!--[if gte IE 5]>
 According to the conditional comment this is Internet Explorer 5 and
up
<![endif]-->
<!--[if lt IE 6]>
 According to the conditional comment this is Internet Explorer lower
than 6
<![endif]-->
<!--[if lte IE 5.5]>
 According to the conditional comment this is Internet Explorer lower
or equal to 5.5
<![endif]-->
<!--[if gt IE 6]>
 According to the conditional comment this is Internet Explorer greate
r than 6
<![endif]-->
```

So, to load stylesheets to specific versions of Internet Explorer in our template we do something like the following:

```
<html>
   <head>
```

```
    ... other CSS files ...
    <!--[if IE 7]>
      <link rel="stylesheet" type="text/css" media="screen" href="{Tem
platePath}/{TemplateName}/css/ie7.css" />
    <![endif]-->
    <!--[if lte IE 6]>
      <link rel="stylesheet" type="text/css" media="screen" href="{Tem
platePath}/{TemplateName}/css/ie6.css" />
    <![endif]-->
  </head>
  ...
</html>
```

Note: Conditional comments used CSS for should be placed inside the <head> tag of a template *after* all other CSS have been linked for their affects to properly take place.

Loading From An Extension

Components

Often a component will have a style sheet of its own. Pushing CSS to the template from a component is quite easy and involves only two lines of code.

```
HubzeroDocumentAssets::addComponentStylesheet('com_example');
```

First, we load the HubzeroDocumentAssets class. Next we call the static method addComponentStylesheet, passing it the name of the component as the first (and only) argument. This will first check for the presence of the style sheet in the active template's [overrides](). If found, the path to the overridden style sheet will be added to the array of style sheets the template needs to include in the <head>. If no override is found, the code then checks for the existence of the CSS in the component's directory. Once again, if found, it gets pushed to the template.

Modules

Loading CSS from a module works virtually the same as loading from a component save one minor difference in code. Instead of calling the addComponentStylesheet method, we call the

addModuleStylesheet method and pass it the name of the module.

```
HubzeroDocumentAssets::addModuleStylesheet('mod_example');
```

Plugins

Loading CSS from a plugin works similarly to loading from a component or module but instead we call the addPluginStylesheet method and pass it the name of the plugin group **and** the name of the plugin.

```
HubzeroDocumentAssets::addPluginStylesheet('examples', 'test');
```

Plugin CSS must be named the same as the plugin and located within a directory of the same name as the plugin inside the plugin group directory.

```
/plugins
  /examples
    /test
      test.css
    test.php
    test.xml
```

View Helpers (all extensions)

Modules, Component, and plugin views now have helpers for pushing Cascading StyleSheets and JavaScript assets to the document. Each method automatically looks for overrides within the current, active template, taking out the busy work of checking yourself each time assets are added. The method names are short, accept a range of options, and allow for method chaining, all tailored for brevity and ease of use.

The css() method provides a quick and convenient way to attach stylesheets. For components, 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 component or plugin will be used. For instance, if called within a view of the component "com_tags", the system will look for a stylesheet named "tags.css".
2. The name of the extension to look for the stylesheet. For components, this will be the

component name (e.g., com_tags). For plugins, this is the name of the plugin folder and requires the third argument of plugin group (type) be passed to the method.
3. *Plugin views only.* The name of the plugin.

Example:

```php
<?php
// Push a stylesheet to the document
$this->css()
     ->css('another')  // Extension (.css) is optional
     ->css('tags.css', 'com_tags');  // Load CSS from another compone
nt
?>
... view HTML ...
```

Along with file names, the method also accepts style declarations:

```php
<?php
// Push a stylesheet to the document
$this->css('.foo {
 color: #000;
}');
?>
... view HTML ...
```

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

And, just as the css() method accepts style declarations, the js() method accepts script declarations:

```php
<?php
// Push some javascript to the document
$this->js('
 jQuery(document).ready(function($){
  $("a").on("click", function(e){
   console.log($(this).attr("href"));
  });
 });
');
?>
... view HTML ...
```

Further Help

Resources for learning and sharpening CSS skills:

- [CSS Zen Garden](#)
- [CSS From The Ground Up](#)
- [Guide to Cascading StyleSheets](#)
- [CSS School](#)

JavaScript

Overview

HUBzero comes with the [jQuery](#) Javascript Framework included by a system plugin. jQuery is not only a visual effects library–it also support Ajax request and JSON notation, table sort, drag & drop operations and much more. All current HUBzero JavaScripts are built on this framework.

Note: If running extensions that require the MooTools Javascript framework (Joomla's default), the jQuery system plugin can be run in compatibility mode or turned off completely.

Directory & Files

The jQuery framework can be found within the /media/system/js directory. It is a compressed version used for production. An uncompressed version may be found at [jquery.com](#).

```
/hubzero
  /media
    /system
      /js
        jquery.js
```

Most HUBzero templates will include some scripts of their own for basic setup, visual effects, etc. These are generally stored in (but not limited to) a sub-directory, named /js, of the template's main directory.

```
/hubzero
  /media
    /system
      /js
        jquery.fancybox.js
        jquery.fileuploader.js
        jquery.ui.js
```

Of the scripts commonly found in a HUBzero template, hub.js is perhaps the most important and it is strongly encouraged that developers include these files in their template.

hub.js

```javascript
//------------------------------------------------------------
//  Create our namespace
//------------------------------------------------------------
var HUB = HUB || {};
HUB.Base = {};

var alertFallback = true;
if (typeof console === "undefined" || typeof console.log === "undefine
d") {
 console = {};
 console.log = function() {};
}

//------------------------------------------------------------
//  Various functions - encapsulated in HUB namespace
//------------------------------------------------------------
if (!jq) {
 var jq = $;

 $.getDocHeight = function(){
  var D = document;
  return Math.max(Math.max(D.body.scrollHeight, D.documentElement.scro
llHeight), Math.max(D.body.offsetHeight, D.documentElement.offsetHeigh
t), Math.max(D.body.clientHeight, D.documentElement.clientHeight));
 };
} else {
 jq.getDocHeight = function(){
  var D = document;
  return Math.max(Math.max(D.body.scrollHeight, D.documentElement.scro
llHeight), Math.max(D.body.offsetHeight, D.documentElement.offsetHeigh
t), Math.max(D.body.clientHeight, D.documentElement.clientHeight));
 };
}

var template = {};

jQuery(document).ready(function(jq){
 var $ = jq,
  w = 760,
  h = 520,
  templatepath = '/templates/{template}/';

 // Set focus on username field for login form
 if ($('#username').length > 0) {
```

```
 $('#username').focus();
}

// Turn links with specific classes into popups
$('a').each(function(i, trigger) {
 if ($(trigger).is('.demo, .popinfo, .popup, .breeze')) {
   $(trigger).on('click', function (e) {
    e.preventDefault();

    if ($(this).attr('class')) {
     var sizeString = $(this).attr('class').split(' ').pop();
     if (sizeString && sizeString.match(/d+xd+/)) {
      var sizeTokens = sizeString.split('x');
      w = parseInt(sizeTokens[0]);
      h = parseInt(sizeTokens[1]);
     }
     else if(sizeString && sizeString == 'fullxfull')
     {
      w = screen.width;
      h = screen.height;
     }
    }

    window.open($(this).attr('href'), 'popup', 'resizable=1,scrollbars
=1,height='+ h + ',width=' + w);
   });
  }
  if ($(trigger).attr('rel') && $(trigger).attr('rel').indexOf('extern
al') !=- 1) {
   $(trigger).attr('target', '_blank');
  }
});

if (jQuery.fancybox) {
 // Set the overlay trigger for launch tool links
 $('.launchtool').on('click', function(e) {
  $.fancybox({
   closeBtn: false,
   href: templatepath + 'images/anim/circling-ball-loading.gif'
  });
 });

 // Set overlays for lightboxed elements
 $('a[rel=lightbox]').fancybox();
}
```

```javascript
  // Init tooltips
  if (jQuery.ui && jQuery.ui.tooltip) {
   $(document).tooltip({
     items: '.hasTip, .tooltips',
     position: {
      my: 'center bottom',
      at: 'center top'
     },
     // When moving between hovering over many elements quickly, the too
ltip will jump around
     // because it can't start animating the fade in of the new tip unti
l the old tip is
     // done. Solution is to disable one of the animations.
     hide: false,
     content: function () {
      var tip = $(this),
        tipText = tip.attr('title');

      if (tipText.indexOf('::') != -1) {
       var parts = tipText.split('::');
       tip.attr('title', parts[1]);
      }
      return $(this).attr('title');
     },
     tooltipClass: 'tooltip'
   });

   // Init fixed position DOM: tooltips
   $('.fixedToolTip').tooltip({
    relative: true
   });
  }

  //test for placeholder support
  var test = document.createElement('input'),
   placeholder_supported = ('placeholder' in test);

  //if we dont have placeholder support mimic it with focus and blur ev
ents
  if (!placeholder_supported) {
   $('input[type=text]:not(.no-legacy-placeholder-
support)').each(function(i, el) {
     var placeholderText = $(el).attr('placeholder');

     //make sure we have placeholder text
     if (placeholderText != '' && placeholderText != null) {
```

```
    //add plceholder text and class
    if ($(el).val() == '') {
      $(el).addClass('placeholder-support').val(placeholderText);
    }

    //attach event listeners to input
    $(el)
      .on('focus', function() {
        if ($(el).val() == placeholderText) {
          $(el).removeClass('placeholder-support').val('');
        }
      })
      .on('blur', function(){
        if ($(el).val() == '') {
          $(el).addClass('placeholder-support').val(placeholderText);
        }
      });
  }
  });

  $('form').on('submit', function(event){
    $('.placeholder-support').each(function (i, el) {
      $(this).val('');
    });
  });
  }
};
```

HUB Namespace

Typically the template will include a file (hub.js) that first establishes a HUB namespace and then proceeds through some basic setup routines. All HUBzero built components, modules, and templates that employ JavaScript place scripts within this HUB namespace. This helps prevent any naming collisions with third-party libraries. While it is recommended that any scripts you may add to your code is also placed within the HUB namespace, it is not required.

Note: When not using jQuery, the template will include a global.js file that establishes the HUB namespace.

Some additional sub-spaces for further organization are available within the HUB namespace. Separate spaces for Modules, Components, and Plugins are created. Once again, this further helps avoid possible naming/script collisions. Additionally, one more Base space is created for basic setup and utilities that may be used in other scripts.

```
//   Create our namespace
if (!HUB) {
 var HUB = {};

 // Establish a space for setup/init and utilities
 HUB.Base = {};

 // Establish sub-spaces for the various extensions
 HUB.Components = {};
 HUB.Modules = {};
 HUB.Plugins = {};
}
```

To demonstrate adding code to the namespace, below is code from a script in a component named com_example.

```
//   Create our namespace
if (!HUB) {
 var HUB = {};

 // sub-space for components
 HUB.Components = {};
}

// The Example namespace and init method
HUB.Components.Example = {
 init: function() {
   // do something
 }
}

// Initialize the code (jQuery)
jQuery(document).ready(function($){
 Components.Example.init();
});
```

Loading From An Extension

Components

Occasionally a component will have scripts of its own. Pushing JavaScript to the template from

a component is quite easy and involves only a few lines of code.

```
HubzeroDocumentAssets::addComponentScript('com_example');
```

First, we load the HubzeroDocumentAssets class. Next we call the static method addComponentScript, passing it the name of the component as the first (and only) argument. This will first check for the presence of the style sheet in the active template's [overrides](#). If found, the path to the overridden script will be added to the array of scripts the template needs to include in the <head>. If no override is found, the code then checks for the existence of the script in the component's directory. Once again, if found, it gets pushed to the template.

Modules

Loading Javascript from a module works virtually the same as loading from a component save one minor difference in code. Instead of calling the addComponentScript method, we call the addModuleScript method and pass it the name of the module.

```
HubzeroDocumentAssets::addModuleScript('mod_example');
```

Plugins

Loading Javascript from a plugin works similarly to loading from a component or module but instead we call the addPluginScript method and pass it the name of the plugin group **and** the name of the plugin.

```
HubzeroDocumentAssets::addPluginScript('examples', 'test');
```

Plugin Javascript must be named the same as the plugin and located within a directory of the same name as the plugin inside the plugin group directory.

```
/plugins
  /examples
    /test
      test.css
    test.php
    test.xml
```

View Helpers (all extensions)

Modules, Component, and plugin views now have helpers for pushing Cascading StyleSheets and JavaScript assets to the document. Each method automatically looks for overrides within the current, active template, taking out the busy work of checking yourself each time assets are added. The method names are short, accept a range of options, and allow for method chaining, all tailored for brevity and ease of use.

The css() method provides a quick and convenient way to attach stylesheets. For components, 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 component or plugin will be used. For instance, if called within a view of the component "com_tags", the system will look for a stylesheet named "tags.css".
2. The name of the extension to look for the stylesheet. For components, this will be the component name (e.g., com_tags). For plugins, this is the name of the plugin folder and requires the third argument of plugin group (type) be passed to the method.
3. *Plugin views only.* The name of the plugin.

Example:

```php
<?php
// Push a stylesheet to the document
$this->css()
     ->css('another')  // Extension (.css) is optional
     ->css('tags.css', 'com_tags');  // Load CSS from another compone
nt
?>
... view HTML ...
```

Along with file names, the method also accepts style declarations:

```php
<?php
// Push a stylesheet to the document
$this->css('.foo {
 color: #000;
}');
?>
... view HTML ...
```

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

And, just as the css() method accepts style declarations, the js() method accepts script declarations:

```php
<?php
// Push some javascript to the document
$this->js('
 jQuery(document).ready(function($){
  $("a").on("click", function(e){
   console.log($(this).attr("href"));
  });
 });
');
?>
... view HTML ...
```

Output Overrides

Overview

There are many competing requirements for web designers ranging from accessibility to legislative to personal preferences. Rather than trying to over-parameterise views, or trying to aim for some sort of line of best fit, or worse, sticking its head in the sand, the CMS gives the potential for the designer to take over control of virtually all of the output that is generated.

Except for files that are provided in the distribution itself, these methods for customization eliminate the need for designers and developers to "hack" core files that could change when the site is updated to a new version. Because they are contained within the template, they can be deployed to the Web site without having to worry about changes being accidentally overwritten when your System Administrator upgrades the site.

HUBzero allows for overriding not only views but CSS and Javascript as well. This allows for even more individualistic styling of components and modules on HUBs.

Component Overrides

Note: Not all HUBzero modules will have layouts or CSS that can be overridden.

Layouts

Layout overrides only work within the active template and are located under the /html/ directory in the template. For example, the overrides for "corenil" are located under /templates/corenil/html/.

It is important to understand that if you create overrides in one template, they will not be available in other templates.

The layout overrides must be placed in particular way. Using "hubbasic2013" as an example you will see the following structure:

```
/templates
.. /kimera
.. .. /html
.. .. .. /com_content  (this directory matches the component directory
 name)
.. .. .. .. /articles       (this directory matches the view director
y name)
.. .. .. .. .. default.php (this file matches the layout file name)
.. .. .. .. .. form.php
```

The structure for component overrides is quite simple:
/html/com_{ComponentName}/{ViewName}/{LayoutName}.php.

Sub-Layouts

In some views you will see that some of the layouts have a group of files that start with the same name. The category view has an example of this. The blog layout actually has three parts: the main layout file blog.php and two sub-layout files, blog_item.php and blog_links.php. You can see where these sub-layouts are loaded in the blog.php file using the loadTemplate method, for example:

```
echo $this->loadTemplate('item');
// or
echo $this->loadTemplate('links');
```

When loading sub-layouts, the view already knows what layout you are in, so you don't have to provide the prefix (that is, you load just 'item', not 'blog_item').

What is important to note here is that it is possible to override just a sub-layout without copying the whole set of files. For example, if you were happy with the Joomla! default output for the blog layout, but just wanted to customize the item sub-layout, you could just copy:

```
/components/com_content/views/category/tmpl/blog_item.php
```

to:

```
/templates/kimera/html/com_content/category/blog_item.php
```

When Joomla! is parsing the view, it will automatically know to load blog.php from com_content natively and blog_item.php from your template overrides.

Cascading Style Sheets

Over-ridding CSS is a little more straight-forward over-ridding layouts. Take the com_groups component for example:

```
/components
  /com_groups
    ...
    com_groups.css    (the component CSS file)
```

To override the CSS, we simply copy or create a new CSS file named the same and place it in the template's overrides:

```
/templates
.. /corenil
.. .. /html
.. .. .. /com_groups   (this directory matches the component directory
 name)
.. .. .. .. groups.css   (this file matches the CSS file name)
```

To push CSS from a component to the template, add the following somewhere in the component:

```
$this->css('example.css');
```

Module Overrides

Note: Not all HUBzero modules will have layouts or CSS that can be overridden.

Layouts

Modules, like components, are set up in a particular directory structure.

```
/modules
.. /mod_latest_news
.. .. /tmpl
.. .. .. default.php   (the layout)
.. .. .. helper.php   (a helper file containing data logic)
.. .. mod_latest_news.php   (the main module file)
.. .. mod_latest_news.xml   (the installation XML file)
```

Similar to components, under the main module directory (in the example, mod_latest_news) there is a /tmpl/ directory. There is usually only one layout file but depending on who wrote the module, and how it is written, there could be more.

As for components, the layout override for a module must be placed in particular way. Using "corenil" as an example again, you will see the following structure:

```
/templates
.. /corenil
.. .. /html
.. .. .. /mod_latest_news    (this directory matches the module directo
ry name)
.. .. .. .. default.php   (this file matches the layout file name)
```

Take care with overriding module layout because there are a number of different ways that modules can or have been designed so you need to treat each one individually.

Cascading Style Sheets

Over-ridding CSS files works in precisely the same way as over-ridding layouts. Take the mod_reportproblems module for example:

```
/modules
  /mod_reportproblems
    ...
    mod_reportproblems.css   (the module CSS file)
```

To override the CSS, we simply copy or create a new CSS file named the same and place it in the template's overrides:

```
/templates
.. /corenil
.. .. /html
.. .. .. /mod_reportproblems   (this directory matches the module dire
ctory name)
.. .. .. .. mod_reportproblems.css
    (this file matches the CSS file name)
```

To push CSS from a module to the template, add the following somewhere in the module:

```
$this->css('mod_example.css');
```

Plugin Overrides

Note: Not all HUBzero plugins will have layouts or CSS that can be overridden.

Layouts

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.

As with components and modules, the layout override for a plugin must be placed in a particular way. Using "corenil" as an example again, you will see the following structure:

```
/templates
.. /corenil
.. .. /html
.. .. .. /plg_groups_forum   (this directory follows the naming patter
n of plg_{group}_{plugin})
.. .. .. .. /browse   (this file matches the layout directory name)
.. .. .. .. .. default.php   (this file matches the layout file name)
```

Take care with overriding plugin layout because there are a number of different ways that plugins can or have been designed so you need to treat each one individually.

Cascading Style Sheets

Over-ridding CSS files works in precisely the same way as over-ridding layouts. Take the forum plugin for groups for example:

```
/plugins
.. /groups
.. .. /forum
.. .. .. /assets
.. .. .. .. /css
.. .. .. .. .. forum.css    (the plugin CSS file)
```

To override the CSS, we simply copy or create a new CSS file named the same and place it in the template's overrides:

```
/templates
.. /corenil
.. .. /html
.. .. .. /plg_groups_forum    (this directory follows the naming pattern of plg_{group}_{plugin})
.. .. .. .. forum.css   (this file matches the CSS file name)
```

To push CSS from a module to the template, add the following somewhere in the module:

```
$this->css('forum.css');
```

Pagination Links Overrides

This override can control the display of items-per-page and the pagination links that are used with lists of information. Most HUBzero templates will come with a pagination override that outputs what we feel is a good standard for displaying pagination links and controls. However, feel free to alter this as you see fit. The override can be found here:

```
/templates/{TemplateName}/html/pagination.php
```

When the pagination list is required, Joomla! will look for this file in the default templates. If it is found it will be loaded and the display functions it contains will be used. There are four functions that can be used:

pagination_list_footer
> This function is responsible for showing the select list for the number of items to display per page.

pagination_list_render
> This function is responsible for showing the list of page number links as well at the Start, End, Previous and Next links.

pagination_item_active
> This function displays the links to other page numbers other than the "current" page.

pagination_item_inactive
> This function displays the current page number, usually not hyperlinked.

Quick Reference

Using the corenil template as an example, here is a brief summary of the principles that have been discussed.

Note: Not all HUBzero components, plugins, and modules will have layouts that can be overridden.

Component Output

To override a component layout (for example the default layout in the article view), copy:

```
/components/com_content/views/article/tmpl/default.php
```

to:

```
/templates/corenil/html/com_content/article/default.php
```

To override a component CSS (for example the stylesheet in the com_groups), copy:

```
/components/com_groups/com_groups.css
```

to:

```
/templates/corenil/html/com_groups/com_groups.css
```

To push CSS from a component to the template, add the following somewhere in the component:

```
ximport('Hubzero_Document');
Hubzero_Document::addComponentStylesheet('com_example');
```

Module Output

To override a module layout (for example the Latest News module), copy:

```
/modules/mod_latest_news/tmpl/default.php
```

to:

```
/templates/corenil/html/mod_latest_news/default.php
```

To override a module CSS (for example the stylesheet in the mod_reportproblems), copy:

```
/modules/mod_reportproblems/mod_reportproblems.css
```

to:

```
/templates/corenil/html/mod_reportproblems/mod_reportproblems.css
```

To push CSS from a module to the template, add the following somewhere in the module:

```
ximport('Hubzero_Document');
Hubzero_Document::addModuleStylesheet('mod_example');
```

Plugin Output

To override a plugin layout (for example the Forum plugin for groups), copy:

```
/plugins/groups/forum/views/browse/tmpl/default.php
```

to:

```
/templates/corenil/html/plg_groups_forum/browse/default.php
```

To override a plugin CSS (for example the stylesheet for the forum plugin for groups), copy:

```
/plugins/groups/forum/forum.css
```

to:

```
/templates/corenil/html/plg_groups_forum/forum.css
```

To push CSS from a plugin to the template, add the following somewhere in the plugin:

```
ximport('Hubzero_Document');
Hubzero_Document::addPluginStylesheet('groups', 'forum');
```

Customise the Pagination Links

To customize the way the items-per-page selector and pagination links display, edit the following file:

`/templates/corenil/html/pagination.php`

Packaging

Preparation

File Structure

The most basic files, such as index.php, error.php, templateDetails.xml, template_thumbnail.png, favicon.ico should be placed directly in your template folder. The most common is to place images, CSS files, JavaScript files etc in separate folders. Joomla! override files must be placed in folders in the folder "html".

```
/{TemplateName}
  /css
    ... CSS files ...
  /html
    ... Overrides ...
  /images
    ... Image files ...
  /js
    ... JavaScript files ...
  error.php
  index.php
  templateDetails.xml
  template_thumbnail.png
  favicon.ico
```

Thumbnail Preview Image

A thumbnail preview image named template_thumbnail should be included in your template. Image size is 206 pixels in width and 150 pixels high. Recommended file format is PNG.

Packaging

Packaging a template 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 the appropriate language sub-directory of /language/ and is loaded each time the template is loaded. All of the other files are copied to the /templates/{TemplateName} subfolder of the HUB installation.

Note to Mac OS X users

The Finder's "compress" menu item produces a usable ZIP format package, but with one catch.

It stores the files in [AppleDouble](#) format, adding extra files with names beginning with "`._`". Thus it adds a file named "`._templateDetails.xml`, which Joomla 1.5.x can sometimes misinterpret. The symptom is an error message, "XML Parsing Error at 1:1. Error 4: Empty document". The workaround is to compress from the command line, and set a shell environment variable "COPYFILE_DISABLE" to "true" before using "compress" or "tar". See the [AppleDouble](#) article for more information.

To set an environment variable on a Mac, open a terminal window and type:

```
export COPYFILE_DISABLE=true
```

Then in the same terminal window, change directories into where your template files reside and issue the zip command. For instance, if your template files have been built in a folder in your personal directory called myTemplate, then you would do the following:

```
cd myTemplate
zip -r myTemplate.zip *
```

Manifest

This XML file just lines out basic information about the template such as the owner, version, etc. for identification by the installer and then provides optional parameters which may be set in the Template 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 template manifest:

```xml
<?xml version="1.0" encoding="utf-8"?>
<extension version="1.5" type="template">
 <name>mynewtemplate</name>
 <creationDate>2008-05-01</creationDate>
 <author>John Doe</author>
 <authorEmail>john@example.com</authorEmail>
 <authorUrl>http://www.example.com</authorUrl>
 <copyright>John Doe 2008</copyright>
 <license>GNU/GPL</license>
 <version>1.0.2</version>
 <description>My New Template</description>
```

```
 <files>
  <filename>index.php</filename>
  <filename>component.php</filename>
  <filename>templateDetails.xml</filename>
  <filename>template_thumbnail.png</filename>
  <filename>images/background.png</filename>
  <filename>css/style.css</filename>
 </files>
 <positions>
  <position>breadcrumb</position>
  <position>left</position>
  <position>right</position>
  <position>top</position>
  <position>user1</position>
  <position>user2</position>
  <position>user3</position>
  <position>user4</position>
  <position>footer</position>
 </positions>
</extension>
```

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

EXTENSION
The install tag has several key attributes. The type must be "template".
NAME
You can name the templates in any way you wish.
FILES
The files tag includes all of the files that will will be installed with the template.
POSITIONS
The module positions used in the template.

The one noticeable difference between this template manifest and the typical manifest of a module or component is the lack of config. While templates may have their own params for further configuration via the administrative back-end, they aren't as commonly found as in other extension manifests. Most HUBzero templates do not include them.

Socicons

Overview

In a single collection, Socicons is a pictographic language containing icons for some of the most popular social and web services such as Twitter, Facebook, and Google.

Integration

The [open source](#) package contains several bootstrap CSS files and fonts for inclusion in your template. Below is the necessary @font-face rules to start using Socicons.

```
@font-face {
 font-family: 'Socicons';
 src: url('/media/system/css/fonts/socicons-webfont.eot');
 src: url('/media/system/css/fonts/socicons-
webfont.eot?#iefix') format('embedded-opentype'),
    url('/media/system/css/fonts/socicons-
webfont.woff') format('woff'),
    url('/media/system/css/fonts/socicons-
webfont.ttf') format('truetype'),
    url('/media/system/css/fonts/socicons-
webfont.svg#SociconsRegular') format('svg');
 font-weight: normal;
 font-style: normal;
}
```

Socicons is relatively lightweight due to the limited number of icons available and can be either included in the stylesheet into your site template or on a per use basis (e.g., individual components).

Use

There are two primary ways to use the font, both with advantages and disadvantages. The first, is to include the necessary HTML and unicode character directly into your markup.

The HTML:

```
<a href="#"><span class="facebook">&#xf013;</span> facebook</a>
```

The CSS:

```
.facebook {
    font-family: "Socicons"
}
```

The advantage here is greater browser compatibility. @font-face is supported by even Internet Explorer 6. The disadvantage, however, is that you now have to edit the HTML wherever you wish to insert an icon which could change depending upon the styling and theme of your template. That could quickly become a headache!

The alternative is to use the CSS pseudo-elements :before and :after. This takes a little more setup in your styles but offers greater flexibility and ease of change. Unfortunately, pseudo-elements are **not** supported in Internet Explorer 7 or older. There is, however, a solution which we'll get to in a moment.

The HTML:

```
<a class="facebook" href="#">facebook</a>
```

The CSS:

```
/* Note the :before pseudo-element */
small.facebook,  /* for IE 7, more on that below */
.facebook:before {
    font-family: "Socicons"
    content: "\\f013"; /* unicode characters must start with a backsla
sh */
}
```

What about Internet Explorer 7?

```
.facebook {
    *zoom:expression(this.runtimeStyle['zoom']='1', this.innerHTML='<s
mall class="facebook">&#xf013;</small>' + this.innerHTML);
```

}

We use <small> in the example above since it's a relatively unused tag and lessens the potential for styling conflicts. It should be noted that over-use of this technique can slow down IE 7 as it has to process and dynamically include content into the page upon render.

Icon List

- \\f002 Hub
- \\f001 Hub alt
- \\f006 Purdue
- \\f005 Purdue alt
- \\f013 Facebook
- \\f012 Facebook alt
- \\f026 Dropbox
- \\f025 Dropbox alt

- \\f011 Twitter
- \\f010 Twitter alt
- \\f019 Github
- \\f018 Github alt
- \\f024 PayPal
- \\f023 PayPal alt
- \\f02a eBay
- \\f029 eBay alt

- \\f017 LinkedIn
- \\f016 LinkedIn alt
- \\f01b Pinterest
- \\f01a Pinterest alt
- \\f022 Skype
- \\f021 Skype alt
- \\f028 Dribbble
- \\f027 Dribbble alt

- \\f02c Google
- \\f02b Google alt
- \\f015 Google+
- \\f014 Google+ alt
- \\f01d Vimeo
- \\f01e Vimeo alt
- \\f01f YouTube
- \\f01e YouTube alt

Fontcons

Overview

In a single collection, Fontcons is a pictographic language designed for a full array of web-related actions and content. Although originally inspired by [Font Awesome](#), we've heavily modified and added to the available icons; Fontcons brings over 250 icons for use in a package equivalent in file size to just one or two bitmapped icons!

Integration

The [open source](#) package contains several bootstrap CSS files for inclusion in your template. These stylesheets can be found in the web root's /media/system/css directory. Here, our attention is on `fontcons.css` which contains the necessary @font-face rules to start using Fontcons.

```
@font-face {
 font-family: 'Fontcons';
 src: url('/media/system/css/fonts/fontcons-webfont.eot');
 src: url('/media/system/css/fonts/fontcons-
webfont.eot?#iefix') format('embedded-opentype'),
   url('/media/system/css/fonts/fontcons-
webfont.woff') format('woff'),
   url('/media/system/css/fonts/fontcons-
webfont.ttf') format('truetype'),
   url('/media/system/css/fonts/fontcons-
webfont.svg#FontconsRegular') format('svg');
 font-weight: normal;
 font-style: normal;
}
```

While you can include Fontcons on a per use basis (e.g., individual components), due to it being relatively light-weight and several Hubzero components making use of it, we recommend including the stylesheet into your site template.

In the <head> of your template's html, reference the location to fontcons.css:

```
<link rel="stylesheet" href="/media/system/css/fontcons.css" />
```

Or import fontcons.css into your site's CSS:

```
/* Note: import rules MUST come first */
@import "/media/system/css/fontcons.css";

/* Other styles here */
```

A word of caution on using @import: Internet Explorer 8 and older will download stylesheets in sequence rather than in parallel. This can have effects on page speed and flashes of un-styled content before the CSS files have finished downloading. See Steve Souder's ["donâ€™t use @import"](#) for more details.

Use

There are two primary ways to use the font, both with advantages and disadvantages. The first, is to include the necessary HTML and unicode character directly into your markup.

The HTML:

```
<a href="#"><span class="edit">&#x270E;</span> edit</a>
```

The CSS:

```
.edit {
    font-family: "Fontcons"
}
```

The advantage here is greater browser compatibility. @font-face is supported by even Internet Explorer 6. The disadvantage, however, is that you now have to edit the HTML wherever you wish to insert an icon which could change depending upon the styling and theme of your template. That could quickly become a headache!

The alternative is to use the CSS pseudo-elements :before and :after. This takes a little more setup in your styles but offers greater flexibility and ease of change. Unfortunately, pseudo-elements are **not** supported in Internet Explorer 7 or older. There is, however, a solution which we'll get to in a moment.

The HTML:

```
<a class="edit" href="#">edit</a>
```

The CSS:

```
/* Note the :before pseudo-element */
small.edit,  /* for IE 7, more on that below */
.edit:before {
    font-family: "Fontcons"
    content: "\\270E"; /* unicode characters must start with a backsla
sh */
}
```

What about Internet Explorer 7?

```
.edit {
    *zoom:expression(this.runtimeStyle['zoom']='1', this.innerHTML='<s
mall class="edit">&#x270E;</small>' + this.innerHTML);
}
```

We use <small> in the example above since it's a relatively unused tag and lessens the potential for styling conflicts. It should be noted that over-use of this technique can slow down IE 7 as it has to process and dynamically include content into the page upon render.

Icon List

- \\f000
- \\266B
- \\f002
- \\2709
- \\2665
- \\2605
- \\2606
- \\f007
- \\f008

WEB DEVELOPERS

- \\f009
- \\f00a
- \\f00b
- \\2714
- \\2716
- \\f00e
- \\f010
- \\f011
- \\f012
- \\2699
- \\f014
- \\2302
- \\f016
- \\f017
- \\2641
- \\f01e
- \\f018
- \\f019
- \\f01a
- \\f01b
- \\f01c
- \\f01d
- \\21BB
- \\f083
- \\f092
- \\f085
- \\f08e
- \\f08d
- \\f077
- \\23F0
- \\f071
- \\f081
- \\260E
- \\f056
- \\f067
- \\f062
- \\f044
- \\f061
- \\f069
- \\f07f
- \\f01f
- \\269B
- \\f09c
- \\f095
- \\f0a1
- \\f0a2

- \\f0a3
- \\f0ad
- \\f0ae
- \\f0b0
- \\f0b2
- \\f0e3
- \\f0d0
- \\f0ea

- \\f021
- \\f022
- \\f023
- \\2691
- \\f025
- \\f026
- \\f027
- \\f028
- \\f029
- \\f02a
- \\f02b
- \\f02c
- \\f02d
- \\f02e
- \\2399
- \\f030
- \\f031
- \\f032
- \\f033
- \\f034
- \\f035
- \\f036
- \\f037
- \\f038
- \\f039
- \\f03a
- \\f03b
- \\f03c
- \\f03d
- \\f03e
- \\f082
- \\2692
- \\25F7

- \\f080
- \\f084
- \\26DF
- \\f004
- \\26D3
- \\f00c
- \\237E
- \\f072
- \\231B
- \\f068
- \\f005
- \\f05c
- \\f054
- \\f063
- \\f053
- \\f07d
- \\f07e
- \\f05f
- \\f09a
- \\f08f
- \\f0a4
- \\f0a5
- \\f0a6
- \\f0a7
- \\f0ca
- \\f0cb
- \\f0cc
- \\f0cd
- \\f0ce
- \\f0db

- \\270E
- \\f041
- \\f043
- \\25D1
- \\270D
- \\f045
- \\2611
- \\f047
- \\21E4
- \\f049
- \\219E

- \\25B6
- \\f04c
- \\2588
- \\21A0
- \\21E4
- \\f049
- \\f052
- \\2039
- \\203A
- \\2295
- \\2296
- \\f057
- \\f058
- \\f059
- \\f05a
- \\f05b
- \\2297
- \\f05d
- \\2298
- \\f087
- \\f088
- \\f086
- \\f091
- \\f093
- \\270B
- \\f00d
- \\f08a
- \\f006
- \\f003
- \\f001
- \\f094
- \\f078
- \\f040
- \\f060
- \\f05e
- \\f08c
- \\f079
- \\f097
- \\f098
- \\f03f
- \\f096
- \\f09d
- \\f0a8
- \\f0a9
- \\f0aa
- \\f0ab

WEB DEVELOPERS

- \\f0b1
- \\f0c1
- \\f0c2
- \\f0c3
- \\2622
- \\2746

- \\2190
- \\2192
- \\2191
- \\2193
- \\f064
- \\f065
- \\f066
- \\271A
- \\2010
- \\273D
- \\f06b
- \\f06c
- \\f06d
- \\2601
- \\f046
- \\f06e
- \\f070
- \\26A0
- \\2757
- \\2708
- \\f073
- \\f074
- \\f075
- \\f0e5
- \\f0e6
- \\f02f
- \\2303
- \\2304
- \\267B
- \\f07a
- \\f07b
- \\f07c
- \\2195
- \\2194
- \\f076

- \\f090
- \\f08b
- \\f089
- \\2661
- \\26A1
- \\2702
- \\22EF
- \\f055
- \\f042
- \\2693
- \\275D
- \\275E
- \\f04a
- \\f048
- \\f04d
- \\f04e
- \\f06f
- \\f04f
- \\f09b
- \\f0a0
- \\f0d7
- \\f0d8
- \\f0d9
- \\f0da
- \\f0d6
- \\f0ea
- \\f0c5

Elements & Typography

Grid (Columns)

For laying out content on a page, the core hub framework includes styles for a 12-column grid.

...

...

...

...

...

...

...

...

...

...

...

...

The grid supports up to 12 columns with span# and offset# classes.

Each column **must** have a .col class. The last column in a set must have the .omega class added for IE 7 to work properly. No clearing div is required.

For example, a four column grid would look like:

```
<div class="grid">
 <div class="col span3">
  ...
 </div>
 <div class="col span3">
  ...
```

```
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

...

Spanning Columns

Columns can be spanned to easier portion content on the page. In the following example, we span the first 6 columns in a container, then follow with two, smaller 3 column containers for a 3-column layout where the first column takes up 50% of the space.

```
<div class="grid">
 <div class="col span6">
  ...
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

Offsets

Columns may also be offset or 'pushed' over.

```
<div class="grid">
 <div class="col span3 offset3">
  ...
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

Helper Classes

.span-quarter
      Span 3 columns. This is equivalent to .span3
.span-third
      Span 4 columns. This is equivalent to .span4
.span-half
      Span 6 columns. This is equivalent to .span6
.span-two-thirds
      Span 8 columns. This is equivalent to .span8

.span-three-quarters
> Span 9 columns. This is equivalent to .span9

A four column grid with the helper classes:

```
<div class="grid">
 <div class="col span-quarter">
  ...
 </div>
 <div class="col span-quarter">
  ...
 </div>
 <div class="col span-quarter">
  ...
 </div>
 <div class="col span-quarter omega">
  ...
 </div>
</div>
```

There are equivalent .offset- classes as well:

.offset-quarter
> Offset 3 columns. This is equivalent to .offset3

.offset-third
> Offset 4 columns. This is equivalent to .offset4

.offset-half
> Offset 6 columns. This is equivalent to .offset6

.offset-two-thirds
> Offset 8 columns. This is equivalent to .offset8

.offset-three-quarters
> Offset 9 columns. This is equivalent to .offset9

Markup for a four column grid with the offset helper class:

```
<div class="grid">
 <div class="col span-quarter">
  ...
 </div>
 <div class="col offset-quarter span-quarter">
  ...
 </div>
 <div class="col span-quarter omega">
```

```
  ...
 </div>
</div>
```

Output:

...

...

...

Nesting Grids

The following is an example of a 3 column grid nested inside the first column of *another* 3 column grid.

```
<div class="grid">
 <div class="col span6">
  <div class="grid">
   <div class="col span4">
    ...
   </div>
   <div class="col span4">
    ...
   </div>
   <div class="col span4 omega">
    ...
   </div>
  </div>
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

...

...

Notifications

The core framework provides some base styles for alter and notifications.

```
<p class="passed">Success message</p>
```

Success message

```
<p class="info">Info message</p>
```

Info message

```
<p class="help">Help message</p>
```

Help message

```
<p class="warning">Warning message</p>
```

Warning message

```
<p class="error">Error message</p>
```

Error message

Sections & Asides

The majority of hub components have content laid out in a primary content column with secondary navigation or metadata in a smaller side column to the right. This is done by first wrapping the entire content in a div with a class of .section. The content intended for the side column is wrapped in a <div class="aside"> tag. The primary content is wrapped in a <div class="subject"> tag and immediately follows the .aside column.

Note: The .aside column must come first in order for the content to be positioned properly. If, unfortunately, this poses a semantic problem, we recommend using the grid system as a potential alternative.

Using aside & subject differs from the grid system in that the .aside column has a fixed width with the .subject column taking up the available left-over space. In the grid system, **every** column is flexible (uses a percentage of the screen) and cannot have a specified, fixed width.

Example usage:

```
<section class="section">
 <div class="section-inner">
  <div class="aside">
   Side column content ...
  </div>
  <div class="subject">
   Primary content ...
  </div>
 </div>
</section>
```

Buttons

{xhub:include type="stylesheet" filename="/media/system/css/buttons.css"}

States

---

default disabled active

```
<a class="btn" href="#">default</a>

<a class="btn disabled" href="#">disabled</a>

<a class="btn active" href="#">active</a>
```

Size

primary secondary

```
<a class="btn btn-primary" href="#">primary</a>

<a class="btn btn-secondary" href="#">secondary</a>
```

Type

link button

```
<a class="btn" href="#">link</a>

<button class="btn" href="#">button</button>

<input type="submit" class="btn" value="input" />
```

Color

danger warning info success

```
<a class="btn btn-danger" href="#">danger</a>

<a class="btn btn-warning" href="#">warning</a>

<a class="btn btn-info" href="#">info</a>

<a class="btn btn-success" href="#">success</a>
```

Icons

danger

warning

info

success

edit

delete

delete

secondary

```
<a class="btn btn-danger icon-danger" href="#">danger</a>

<a class="btn btn-warning icon-warning" href="#">warning</a>

...
```

Groups

Dropdown

- Action
- Another action
- Something else here
-
- Separated link

```
<div class="btn-group dropdown">
        <a class="btn" href="#">Dropdown</a>
        <span class="btn dropdown-toggle"></span>
        <ul class="dropdown-menu">
          <li><a href="#">Action</a></li>
```

```
            <li><a href="#">Another action</a></li>
            <li><a href="#">Something else here</a></li>
            <li class="divider"></li>
            <li><a href="#">Separated link</a></li>
        </ul>
</div>
```

Dropup

- [Action](#)
- [Another action](#)
- [Something else here](#)
-
- [Separated link](#)

```
<div class="btn-group dropup">
        ...
</div>
```

Dropdown

- [Action](#)
- [Another action](#)
- [Something else here](#)
-
- [Separated link](#)

```
<div class="btn-group btn-secondary dropdown">
        ...
</div>
```

```
<div class="btn-group">
        <a class="btn icon-prev" href="#">prev</a>
        <a class="btn" href="#">all</a>
        <a class="btn icon-next opposite" href="#">next</a>
</div>
```

Muse

Overview

Muse, with its connotation of inspiration and creativity, is the HUBzero framework for command line tools and automation. Muse, by default, has commands for running migrations, clearing the cache, creating scaffolding, updating your hub, and more. In addition to the default commands, Muse can also be extended by individual components to provide component specific tools and command line functionality. We'll walk through many of the detailed commands below, and then give a brief description of how you can add your own commands to Muse.

Note that all commands below are assumed to come from your hub's document root.

So, to run muse, simply type:

```
php muse
```

This, like many other commands, will return your available options by default. The current list of top level commands includes:

- Cache
- Configuration
- Database
- Environment
- Extension
- Group
- Log
- Migration
- Repository
- Scaffolding
- Test
- User

As a developer, you may find yourself in a given moment as either a consumer of existing commands, or a creator of new commands, To understand the existing commands, jump over to the commands chapter for more details about each command. Continue below to learn more about creating your own commands.

Structure

Muse by default will look for commands in the Commands directory within the Hubzero Console Library. If you're looking to add a new core command, this is where it will live.

The name of the command file becomes the name of the command itself. So, for example, the Database command would be found at:

```
core/libraries/Hubzero/Console/Command/Database.php
```

Within the file itself, all public methods will be considered tasks that can be called on the command. Private and protected methods will not be directly routable. To exemplify this, you'll notice that the Database command has two tasks, dump and load. These are public methods with the Database.php command class. At a minimum, commands are to implement the CommandInterface, which requires three methods:

```
public function __construct(Output $output, Arguments $arguments);
public function execute();
public function help();
```

By extending the base command, you can further simplify things to only need the execute and help methods. The execute task is the default task and is called when no task is explicity given. The help command should establish meaninful descriptions of tasks and arguments available.

Often times it will make sense to simply route the execute command to the help command, thus giving users an overview of your command and options by default

You can also namespace your commands. And by this we simply mean that you can use folders to create logical subdivisions within your commands. You'll see this, for example, in the Configuration command. The configuration command has two subcommands, aliases and hooks. To call tasks on these commands, you simply:

```
php muse configuration:hooks add ...
php muse configuration:aliases help
```

Arguments and Output

Within the command there are two primary objects of interest on the command, the arguments and the output.

Arguments

The primary function of the arguments class is to provide the command with access to the extra arguments passed into the command by the user. There are really two primary styles or ways of structuring a command arguments. For required commands, we typically use an ordered variable approach to these arguments. Consider the scaffolding command. It expects a task of the scaffolding action we are to perform, such as create or copy. We then expect the type of item we will be scaffolding. This ultimately will look as follows:

```
php muse scaffolding create migration
```

Then, to access these types of arguments, we simply grab them by their index order:

```
$type = $this->arguments->getOpt(3);
```

The index numbers follow the underlying values from PHP's native arguments, where the script is 0, the command is 1, the task is 2, and so on from there.

In addition to this initial style of argument, you can also accept named arguments. These are often optional sorts of arguments, such as:

```
php muse scaffolding create migration --install-dir=/altlocation
```

And these would be accessed in a similar manner:

```
$installDir = $this->arguments->getOpt('install-dir');
```

Output

Throughout the course of your command, it's important to let the user know what you're doing, and whether or not everything was successful. To do that, we use the output object on the command. The primary methods of interest are:

```
$this->output->addLine('Hello');
```

```
$this->output->addString('hello');
$this->output->error('Something went wrong!');
```

Hopefully the method names are fairly self-explanatory. The addLine method adds the given string along with a newline, whereas the addString simply outputs the given message. The error command outputs the given message with error styling, and also stops execution immediatly (this is important!).

Both the addLine and addString methods accept a second argument specifying a style for the message. Available shortcut strings include: warning, error, info, and success. More fine-grained control can be achieved by passing an array as the second parameter. This array can have up to three arguments, specifying a format, color, and indentation. The available formats include:

- normal
- bold
- underline

And available colors include:

- black
- red
- green
- yellow
- blue
- purple
- cyan
- white

It's important to remember that care should be taken when specifying colors, as a given user's console styles may make reading certain colors more difficult.

Here are some examples of using the message styles:

```
$this->output->addLine('All done here', 'success');
$this->output->addLine('Something went wrong!', ['color' => 'red', 'fo
rmat' => 'bold']);
```

Documentation

Documenting your commands is a good practice, both for you and for those that will be using your commands. All commands are required to have a help function. That function will be used to output the appropriate help info for the command. A typical help method will look something like this:

```php
public function help()
{
    $this
        ->output
        ->addOverview(
            'This is my command for doing great things'
        )
        ->addTasks($this)
        ->addArgument(
            '--awesome-level: Set the awesomeness level',
            'Specify the desired level of awesomeness',
            'Example: --awesome-level=7'
        );
}
```

The methods available for help documentation fairly straight-forward. The overview section, generated by addOverview, is the main description of the command. The addTasks method is used generate a list of available tasks within the command. Finally, the addArgument method can be used to specify the available arguments that your command accepts.

The addTasks method generates the available tasks list based on public methods, as mentioned above. To define the description for the method, include the @museDescription tag in the method docblock, as shown below.

```php
/**
 * Creates awesomeness
 *
 * @museDescription Constructs and does important things
 *
 * @return void
**/
```

The result of the above examples would render like this:

```
me@myhub.org:~# muse mycommand help
Overview:
  This is my command for doing great things

Tasks:
  create    Constructs and does important things

Arguments:
  --awesome-level: Set the awesomeness level
    Specify the desired level of awesomeness
    Example: --awesome-level=7
```

Interactivity

Interactivity is a cool feature of Muse. This allows a more guided experience for users. For example, instead of requiring users to provide four arguments, you can prompt for them, or even tailor them based on previous arguments. An example of this can be found in the extension command.

```
me@myhub.org:~# muse extension
What do you want to do? [add|delete|install|enable|disable] add
What extension were you wanting to add? com_awesome
Successfully added com_awesome!
```

To display a prompt to the user, simply use the getResponse method on the output object.

```
$name = $this->output->getResponse("What extension were you wanting to
 add?");
```

This will wait for a response and enter from the user.

When not to be interactive?

Interactivity is not always desired. If a user has set the non-interactive flag, or the current output mode is non-standard, it becomes important to not wait for user input. To ensure proper functionality in different environments and output formats, you should wrap all interactive calls in

the isInteractive check and provide an appropriate alternative (likely just checking for a given argument).

```
// Check for interactivity
if ($this->output->isInteractive())
{
    // Prompt for action
    $action  = $this->output->getResponse('What do you want to do?');
}
else
{
    // Otherwise show help output so user knows available options
 $this->output = $this->output->getHelpOutput();
 $this->help();
 $this->output->render();
 return;
}
```

Component Commands

In addition to the basic command library, individual commponents can contain commands as well. This makes adding site-specific commands easier (without modifying core HUBzero), as well as allowing for a more logical grouping of functionality with other component-specific models.

Site commands work in exactly the same manner as library commands, but are simply located in an alternate place.

```
app/components/mycomponent/cli/commands/mycommand.php
```

Commands must still implement the command interface, and should function the same way as library commands. They will not however, show up in the master command list obtained when calling the global muse help command

Commands

Cache

The cache command is a helper for clearing your sites cache files. You can clear the entire cache, or just the CSS cache. Those commands, respectively, are:

```
php muse cache clear
```

```
php muse cache:css clear
```

Configuration

The configuration command is used to personalize and customize your Muse experience. It's also used to store variables for repeated use. For example, the scaffolding command will ask you, if you haven't already, to set your name and email to be used when generating files.

```
muse configuration set --user_name="John Doe"
muse configuration set --user_email=john.doe@gmail.com
```

Configuration can also be used to store hooks and aliases. Hooks are additional commands that are run at pre-defined points. Aliases are command shortcuts. Here are some examples:

```
# run permissions fix after updating the repository
muse configuration:hooks add repository.afterUpdate "chmod -R g+w /www
/docroot"

# Add a shortcut for the environment command
muse configuration:aliases add env environment
```

Database

The database command was added for two primary reasons - the first backups, and the second, reverse content migration. Backups are fairly straight-forward, but a little more detail is in order for reverse content migration.

If you have an environment with more than one stop in your production flow, you've likely run into the problem of wanting to move data from prod to dev for testing purposes. But in so doing, you often overwrite some site-specific configuation on dev. So get around this, we perform a dump and load using the database command to move only those things that should move between environments.

```
# dump the database
muse database dump

# then make sure you copy to your dev environment
# then from dev, load the dump back up (it will have a different name)
muse database load filenamefromabovecommand
```

Environment

The environment command simply outputs the current environment variables.

```
Current user     : Mr Awesome <awesome@gmail.com>
Current database : example
```

Extension

If you don't already know, extensions are the general name for all of the 'apps' allowed by the HUBzero framework. They include (amoung some others), templates, components, modules, and plugins. When adding a new extension, you will often want to add it to the extensions database table and enable it. This command can help save you trips directly to the database.

The nice thing to about the extension command is that it will prompt you for what it needs, you don't really need to remember the syntax.

```
me@me.org:~# muse extension
What do you want to do? [add|delete|install|enable|disable] add
What extension were you wanting to add? com_awesome
Successfully added com_awesome!
```

Or, as another example. Let's delete that entry we added above using the written out syntax

```
me@me.org:~# muse extension delete --name=com_awesome
Successfully deleted com_awesome!
```

Note that if you're in a production environment and using migrations, this command is redundent. Use migrations! But if you're just testing and need a quick way to enable or disable something, this is the way to go.

Group

The group commands are simply wrappers on existing commands to be used within the super group context. Please review the super group documentation for more details.

Log

The log command is great for following and filtering log entries. There are currently two log types available, the profile log and the query log. To start, simply:

```
muse log follow profile
```

You have to having logging enabled for new entries to be displayed!

Once started, you'll see info on the log fields being displayed.

```
me@me.org:~# muse log follow profile
The profile log has the following format (* indicates visible field):
<0:*timestamp> <1:*hubname> <2:*ip> <3:*app> <4:*uri> <5:*quer
y> <6:*memory> <7:*querycount> <8:*timeinqueries> <9:*totaltime>
```

To toggle a fields visibility, simply press the number next to the field of interest. For example, pressing 2, and then f to show the fields again, results in:

```
> Hiding ip
> The profile log has the
following format (* indicates visible field):
<0:*timestamp> <1:*hubname> <2:ip> <3:*app> <4:*uri> <5:*quer
y> <6:*memory> <7:*querycount> <8:*timeinqueries> <9:*totaltime>
```

To show the available commands, simply type h.

```
> q: quit, h: help, i: input mode, p: pause/play, b: beep on/off, f: f
ields, r: rerender last 100 lines
```

Migration

For more info on the migration command, see the dedicated [migrations](#) section under the database chapter.

Repository

The repository command offers an abstraction on top of the mechanism used to manage and update the CMS. This could include GIT, HTTP-based package installs, or Debian packages. Currently, GIT is the only supported mechanism, but more are to come in the future.

To start, simply see if the repository command is supported in your environment.

```
me@me.org:~# muse repository
This repository is managed by GIT and is clean
```

If you environment is not currently supported, you'll receive a message like this:

```
me@me.org:~# muse repository
Sorry, this command currently only supports setups managed by GIT
```

To start the update process, use the update task. Depending on your current state, you'll either see that you're up-to-date, or see what's coming in the next update.

```
me@me.org:~# muse repository update
The repository is already up-to-date
```

or...

```
me@me.org:~# muse repository update
The repository is behind by 747 update(s):
...
```

Then, to perform the actual update, add the -f flag.

```
me@me.org:~# muse repository update -f
Updating the repository...complete
```

If something goes wrong, the update mechanism will automatically roll back to it's state prior to attempting the update. Then you'll have to go in a manually perform the update depending on the mechanism.

Spring Cleaning

In addition to performing updates, the repository command also offers some help doing periodic cleanup. Using the clean command will allow you to prune rollback points and stashes.

```
me@me.org:~# muse repository clean
Do you want to purge all rollback points except the latest? [y|n] y
Purging rollback points.
Do you want to purge all stashed changes? [y|n] y
Purging repository stash.
Clean up complete. Performed (2/2) cleanup operations available.
```

Scaffolding

Scaffolding was create to help developers get started quickly. Let's be honest, developers rarely start from a blank file. We copy something existing and modify. With scaffolding, we give you a template a pre-fill known values to make this process even easier.

At this time, scaffolding knows how to create:

- Commands
- Components
- Migrations
- Tests

So, for example, to create a new component, simply:

```
me@me.org:~# muse scaffolding create component com_awesome
Creating /var/www/example/core/components/com_awesome/awesome.xml
Creating /var/www/example/core/components/com_awesome/admin/awesome.php
Creating /var/www/example/core/components/com_awesome/admin/controllers/awesome.php
Creating /var/www/example/core/components/com_awesome/admin/language/en-GB/en-GB.com_awesome.ini
Creating /var/www/example/core/components/com_awesome/admin/language/en-GB/en-GB.com_awesome.sys.ini
Creating /var/www/example/core/components/com_awesome/admin/views/awesome/tmpl/display.php
Creating /var/www/example/core/components/com_awesome/api/controllers/api.php
Creating /var/www/example/core/components/com_awesome/config/access.xml
Creating /var/www/example/core/components/com_awesome/config/config.xml
Creating /var/www/example/core/components/com_awesome/models/awesomes.php
Creating /var/www/example/core/components/com_awesome/site/awesome.php
Creating /var/www/example/core/components/com_awesome/site/assets/css/awesome.css
Creating /var/www/example/core/components/com_awesome/site/assets/js/awesome.js
Creating /var/www/example/core/components/com_awesome/site/controllers/awesome.php
Creating /var/www/example/core/components/com_awesome/site/language/en-GB/en-GB.com_awesome.ini
Creating /var/www/example/core/components/com_awesome/site/router.php
Creating /var/www/example/core/components/com_awesome/site/views/awesomes/tmpl/display.php
Creating /var/www/example/core/components/com_awesome/site/views/aweso
```

```
mes/tmpl/edit.php
```

As you can see, this automatically generates all of the core files and views you're likely to need. It also names them appropriately, as well as using the provided component name to even tweak the contents of these files.

Test

Testing is critical to both deploying a new extension, and updating existing extensions without too much heartache. To facilitate testing, muse offers a framework and wrapper around the popular PHP Unit testing infrustucture.

To see the current extensions with tests, run:

```
me@me.org:~# muse test show
lib_database
```

Then, to run a specific extensions tests, you can use the run command.

```
me@me.org:~# muse test run lib_database
PHPUnit 4.6.2 by Sebastian Bergmann and contributors.

....................................................

Time: 2.26 seconds, Memory: 17.5Mb

OK (51 tests, 73 assertions)
```

User

The final command available at this time is the user command. It offers some advances administrative functionality for merging and unmerging users.

This command is experimental!

Occasionally, on a hub, one person will create two accounts and not realize it. They later ask

you to merge the accounts and move the contributions from one to the other. This isn't a simple task, and involves updating many, many references in the database. Fortunately for you, we've been working on a solution.

```
me@me.org:~# muse user merge 1042 into 1003
Updating (1) item(s) in jos_collections.object_id
Updating (1) item(s) in jos_collections.created_by
Updating (1) item(s) in jos_collections_items.created_by
Updating (8) item(s) in jos_courses_asset_groups.created_by
Updating (15) item(s) in jos_courses_assets.created_by
Updating (1) item(s) in jos_courses_members.user_id
Updating (2) item(s) in jos_courses_offering_section_dates.created_by
Updating (2) item(s) in jos_courses_units.created_by
Updating (76) item(s) in jos_developer_access_tokens.uidNumber
Updating (1) item(s) in jos_developer_applications.created_by
Updating (1) item(s) in jos_developer_rate_limit.uidNumber
Updating (9) item(s) in jos_users_log_auth.user_id
Ignoring jos_users_password.user_id due to integrity constraint violat
ion
Updating (1) item(s) in jos_users_points.uid
Ignoring jos_xprofiles_bio.uidNumber due to integrity constraint viola
tion
Updating (3) item(s) in jos_xprofiles_tokens.user_id
```

Then, if needed, you can reverse the merge.

```
me@me.org:~# muse user unmerge 1042 from 1003
Unmerged (122/122) records successfully!
```

Super Groups

Overview

Super groups are advanced HUB groups, that have their own webspace within the HUB to showcase their group.

Super groups have a lot of extra functionality built in to allow them to customize their group.

Group Pages & Modules

Super groups have the ability to include PHP and javascript code into group pages and modules. Pages or modules that contain PHP or Javascript code will then need to be approved by a group page approver. Notifications are sent to approvers when a page needs to be approved. Another notification will be sent to the group managers when the page has been approved.

Templating System

Overview

A new templating system has been added to help Super groups create a better web presence. When a super group is created, a default template is created and placed in the groups filespace.

The only file needed for a super group template to work is /{web_root}/site/groups/{group_id}/template/index.php

File Structure

Below shows the desired file directory structure for super groups. Following this pattern will allow HUB owners and developers to add new developments and find bugs easier.

Default Template

A default template is created for each super group. This can be used as a base for the super groups template.

Error Template

Super groups have the ability use a custom error template (error.php), which can include a stylesheet (error.css) or scripts to display a custom error page.

Template Includes

The following group include tags can be used within a template to display the content, the menu, the member/manager toolbar, modules, or include a Google Analytics tracking code.

- <group:include type="content" />
- <group:include type="content" scope="before" />
- <group:include type="menu" />
- <group:include type="toolbar" />
- <group:include type="modules" postion="{position}" />
- <group:include type="modules" title="{title}" />
- <group:include type="googleanayltics" account="{account}" />
- <group:include type="script" base="" source="{file_path}" />
- <group:include type="stylesheet" base="" source="{file_path}" />

For Script & Stylesheet group includes you can specify a base param of "template" which will automatically prepend "/template/assets/js" or "/template/assets/css" to the source. If no base is specified, it will look for the file in the groups "uploads" directory.

Page Templates

Overview

You'll probably want most of your group pages to look about the same. Sometimes, though, you may need a specific page, or a group of pages, to display or behave differently. This is easily accomplished with page templates.

Specialized Page Templates

Create a template for one Page: Intended for one specific page, you can create a specialized template, named with that page's alias or ID:

1. page-{alias}.php
2. page-{id}.php

For example: Your About Us page has an alias of 'about-us' and an ID of 6. If template has a file named page-about-us.php or page-6.php, then it will automatically find and use that file to render the About Us page.

To be used, specialized page templates must be in your groups template directory: /{web_root}/site/groups/{group_id}/template/

Custom Page Templates

Create a template that can be used by any page: A custom page template can be used by multiple pages. To create a custom page template make a new file starting with a template name inside a PHP comment. Here's the syntax:

```php
<?php
/*
Template Name: My Custom Page
*/
```

To be used, custom page templates must be in your groups template directory: /{web_root}/site/groups/{group_id}/template/

Selecting a Page Template

Once you upload the file to your template's folder, the template name, "My Custom Page", will list in the edit page screen's Template dropdown.

Template Hierarchy

The order below defines which page template gets loaded on any given page. The first match found is used.

1. Custom Template â€" If the page has a custom template assigned, the HUB will looks for that file and, if found, use it.
2. page-{alias}.php â€" Else the HUB looks for and, if found, uses a specialized template named with the page's alias.
3. page-{id}.php â€" Else the HUB looks for and, if found, uses a specialized template named with the page's ID.
4. page.php â€" Else the HUB looks for and, if found, uses the default page template.
5. index.php â€" Else the HUB uses a the template's index file.

Page Includes

The following group include tags can be used within a group page.

- <group:include type="modules" postion="{position}" />
- <group:include type="modules" title="{title}" />
- <group:include type="script" base="" source="{file_path}" />
- <group:include type="stylesheet" base="" source="{file_path}" />

For Script & Stylesheet group includes you can specify a base param of "template" which will automatically prepend "/template/assets/js" or "/template/assets/css" to the source. If no base is specified, it will look for the file in the groups "uploads" directory.

Custom Macros

Overview

Super groups have the ability to create their own custom macros or override any existing macro [[MacroName(args)]].

Custom Macro Class Structure

```php
<?php
/**
 * HUBzero CMS
 *
 * Copyright 2005-2014 Purdue University. All rights reserved.
 *
 * This file is part of: The HUBzero(R) Platform for Scientific Collaboration
 *
 * The HUBzero(R) Platform for Scientific Collaboration (HUBzero) is free
 * software: you can redistribute it and/or modify it under the terms of
 * the GNU Lesser General Public License as published by the Free Software
 * Foundation, either version 3 of the License, or (at your option) any
 * later version.
 *
 * HUBzero is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program.  If not, see .
 *
 * HUBzero is a registered trademark of Purdue University.
 *
 * @package   hubzero-cms
 * @copyright Copyright 2005-2014 Purdue University. All rights reserved.
 * @license   http://www.gnu.org/licenses/lgpl-3.0.html LGPLv3
 */
```

```
namespace PluginsContentFormathtmlMacros;

use PluginsContentFormathtmlMacro;

/**
 * Wiki macro class for displaying hello world
 */
class {{macro_name}} extends Macro
{
 /**
  * Returns description of macro, use, and accepted arguments
  *
  * @return     array
  */
 public function description()
 {
  $txt = array();
  $txt['html'] = '

Put macro description here...

';
   return $txt['html'];
 }

 /**
  * Generate macro output
  *
  * @return     string
  */
 public function render()
 {
  return 'Return any html content you want this macro to render';
 }
}
```

Overriding Macros

To override a macro, copy the original macro located in
/{web_root}/plugins/content/formathtml/macros/ into your groups macro folder
/{web_root}/site/groups/{group_id}/macros/. You can no modify the render functionality, add or

remove params, etc.

Note: The Macos class name must remain the same and the class must implement a render() method. You are free to add or chanage other methods.

Note: If the original macro file is located within a subfolder, you must recreate that folder structure in the groups macros folder for the override to work.

PHP Pages

Overview

Super groups have the ability to include PHP code in any group page or module through the page and module managers. If you are finding this is hard to manage or the approval process is taking too long. Users with SSH access and PHP knowledge can add any number of PHP pages to their super group.

PHP Pages Directory

/{web_root}/site/groups/{group_id}/pages/

PHP Page Hierarchy

- /{web_root}/site/groups/{group_id}/pages/features.php -> /groups/{group_cn}/features
- /{web_root}/site/groups/{group_id}/pages/features/one.php -> /groups/{group_cn}/features/one
- /{web_root}/site/groups/{group_id}/pages/features/two.php -> /groups/{group_cn}/features/two

PHP Page Includes

The following group include tags can be used within a PHP page.

- <group:include type="modules" postion="{position}" />
- <group:include type="modules" title="{title}" />
- <group:include type="script" base="" source="{file_path}" />
- <group:include type="stylesheet" base="" source="{file_path}" />

For Script & Stylesheet group includes you can specify a base param of "template" which will automatically prepend "/template/assets/js" or "/template/assets/css" to the source. If no base is specified, it will look for the file in the groups "uploads" directory.

Databases

Overview

Each super group comes with its own database. This database can be used to store data for that group. The credentials for accessing that database can be found in the super groups database config file.

Config Path

/{web_root}/site/groups/{group_id}/config/db.php

Config File Contents

```php
<?php
return array(
 'host' => 'localhost',
 'port' => '',
 'user' => 'sgmanager',
 'password' => 'xxxxx',
 'database' => 'sg_{group_cn}',
 'prefix'  => ''
);
```

Using the Database

You can use the database anywhere you want in your template, a PHP page, a group component, etc. Anywhere you can run PHP code basically.

Getting a reference to the group database object is very easy:

```php
$database = HubzeroUserGroupHelper::getDBO();
```

You can access the group database and the HUB database at the same time. Use the above call to get access to the group database and JFactory::getDBO(); to get access to the HUB database. All you have to do is store them in two different variables.

Migrations

Overview

Migrations allow you a group update its separate database without having to connect to the live database and manually updating the schema. Another benefit to using migrations is that they are automatically run every time the super groups code is updated from Gitlab!

Creating a Migration

Migrations can be created easily with the HUbzero command line application "Muse". From the command line run the following command (in the web root):

/{web_root}/cli/muse.php group scaffolding migration --group={group_cn} -e=com_{component}

Simply replate {group_cn} with your groups cname and {component} with the component. The migration file will be automatically placed into the correct location, ready for you to modify and commit when ready.

Running Migrations

Running migrations is almost as easy as creating them with once again help from the Hubzero command line application. From the command line run the following command (in the web root):

/{web_roo}/cli/muse.php group migrate -if --group={group_cn}

Simply replate {group_cn} with your groups cname. The -i argument means ignore dates (run migrations it could have missed) and -f means actually run.

Note: You only need to manually run migrations in a dev environment. When groups code is updated on live, migrations are automatically run.

Components

Overview

Super groups now have the ability to have their own components. They are setup the <u>exact</u> same as a regular component except for their file location. For more information regarding developing components see: <u>https://hubzero.org/documentation/1.3.0/webdevs/components</u>

Components Directory

/{web_root}/site/groups/{group_id}/components/com_{component}/

Component Language Files

/{web_root}/site/groups/{group_id}/language/en-GB/en-GB.com_{component}.ini

Component Paths

As a helper for super group component developers the path to the component directory is defined in a constant.

JPATH_GROUPCOMPONENT

So as an example, if your creating the component "com_drwho", the JPATH_GROUPCOMPONENT constant equals:

/{web_root}/site/groups/{group_id}/components/com_drwho/

Note: You should be able to move the component to the main components folder and and have it work without any changes.

URL's built within a super group component will automatically have "/groups/{group_cn}/" prepended to them. Please don't manually do that in your component or it will result in an error.

Creating a Component

Creating components can always be done manually by creating the files in the correct location as described above. You can also utilize the Hubzero command line application. From the command line run the following command (in the web root):

/{web_root}/cli/muse.php group scaffolding component --group={group_cn}
-n=com_{component}

Simply replate {group_cn} with your groups cname and {component} with the component. The component files will be automatically placed into the correct location, ready for you to modify and commit when ready.

Super Groups & Gitlab

What is Gitlab

In efforts to make super group development easier, we are utilizing a code management tool called [Gitlab](#). Similar to [Github](#) in functionality and looks, it provides an easy way for the developers to write & push code and give the HUBzero team the security we needed to allow third party developers to commit code to production machines.

Why Gitlab

Overview

There are many benefits for both parties (the developer & the Hub team) to using a tool like Gitlab for managing the code of a super group.

Security

Live site access can be very dangerous for even a very experienced developer. The use of Gitlab removes the need to access the live site all together. The developer can code and test in whatever environement they want, add their changes and the hub can pull in the approved changes right through the HUB admin interface.

Gitlab also allows the HUB team to monitor and approve code after the developer has pushed their updates. Changes MUST be approved by the HUB team before they can be moved to the live site.

Developer Freedom

Developers will actually work on whats called a "forked" copy of the super group repository. This means they they are working on their very own version of the super group code and can do whatever they want to it without affecting the live site super group or any other developers also working on the super group.

Frequent Updates

Managing the super groups by Gitlab (outside of the main CMS), allows for more regular updates. At any point after the hub is configured to work with Gitlab, any hub admin who has access to the groups adminstrator panel can update the the super groups code.

Extra Features

Along with a nice code browser/editor, Gitlab comes with an Issue tracker and Wiki section for each project. Each project has the ability to use those sections however these please.

Setup

Hub Setup

Each HUB can choose to integrate with Gitlab or not. If your HUB chooses to integrate then there are few steps to get setup and running.

1. Gitlab integration must be enabled in the Groups config, under the "super" tab. The Gitlab API URL must also be supplied along with the API key of an admin account on Gitlab (found under the "account" tab in profile section in Gitlab). This allows the HUB to do the initial group/project/repository creation when the super group is created.
2. In order for the HUB (www-data user) to make the first commit to the project, including the basic super group template and folder structure, the www-data user must have an SSH Key on the HUB machine. That SSH key must also be added to an admin account on Gitlab. This first commit actually creates the GIT repository in Gitlab.
3. The last step for HUB setup is to SSH as the www-data user to the Gitlab machine from the HUB machine.. This will approve the RSA fingerprint of the Gitlab machine for the www-data user and add the machine to the known_hosts file. If this step is omitted the hubs attempt to make the inital commit will be denied.

Group Setup

When a super group is created on the HUB, most of the initial setup for that super group is done automatically. If you are working on a super group that was created prior to May 2014, then this setup will need to be done manually for that super group to work with Gitlab. Please enter a ticket through the support system detailing the the super group and that you would like to have your group integrated into Gitlab.

Developer Setup

Access

Part of the developer setup is getting permission to access Gitlab. This must be done manually by the HUBzero development team. Please submit a ticket indicating the super group and any users (name, preferred username, & email address) that will need to have access in Gitlab for the project.

Login & Password Change

Once you have submitted a ticket for access to Gitlab, you will recieve an email within 48 hours with all the details you need to login to your account. The email will contain a temporary

password that you will be forced to update upon login. After you login and change your password you can move on to the next step, uploading an SSH key.

SSH Keys

In order to make commits and push to Gitlab, you need to add an SSH key to your account. To add an SSH key, login to Gitlab, go to your profile, then SSH Keys. Click the "Add SSH Key" button, enter any title you want, paste your public SSH key in the box, and click "Add Key". If you are unsure of how to create an SSH key there is a link at the top of the "Add an SSH Key" page that links to a help page with detailed instructions.

Note: You can add multiple SSH keys if you plan to make commits from multiple machines.

Developing

Overview

After you have completed all the necessary setup steps its time to start actually developing. The following items are necessary steps to getting your code added to the live site super group code as easily as possible.

For example purposes we are going to use "mytestgroup" as the group cname and "hubzero.org" as the hub we are working on. This would map to "hubzero" as the group name and "mytestgroup" as the project in Gitlab. We are also going to use "theuser" as the user's username in Gitlab.

Fork Project

The first step is you need to create a fork of the main project. You can find main project by navigating to your dashboard in Gitlab then the projects tab. Project names are formatted by the group/project, where group is the hub name/URL and project is super group cname.

1. Click on the project you want to start development for, you should be taken to the project page.
2. Click the "Fork repository" button on the right side of the page. This will fork the repository and take you to your forked version of this repository.

Clone Repository

You are now ready to clone the repository to a development machine. This can be anywhere, but recommended that you use the hubs dev machine or local dev machine (local HUB on VM).

1. Get the repository url. From your forked repository page you should see a text box with the git repo url in it. Copy that URL to your clipboard
2. Go to the machine where you want to clone the repository to and type the following into a terminal window:

   ```
   git clone git@gitlab.hubzero.org:theuser/mytestgroup.git; mv myte
   stgroup/* mytestgroup/.git* .; rmdir mytestgroup;
   ```

3. The repository content will be copied to a "mytestgroup" directory within the current directory

Add Upstream Repository

Upstream repository is a fancy word for the main repository you forked from. You need to tell your forked copy that it has a main repository and where it is. To add the upstream repository, in a Terminal window navigate to your cloned repo and type the following:

```
git remote add upstream git@gitlab.hubzero.org:hubzero/mytestgroup.git
```

You can test to see if everything was added correctly by typing:

```
git remote -v
```

and you should now see something like:

```
origin    git@gitlab.hubzero.org:testuser/mytestgroup.git (fetch)
origin    git@gitlab.hubzero.org:testuser/mytestgroup.git (push)
upstream  git@gitlab.hubzero.org:hubzero/mytestgroup.git (fetch)
upstream  git@gitlab.hubzero.org:hubzero/mytestgroup.git (push)
```

This is a very important part of working with Gitlab is keeping your forked repository synced with the main repository.

Develop

Make changes, add new code, fix bugs etc. Commit as you develop.

Sync with Main Project

Before you push your changes to Gitlab it is recommended that you sync your forked project with the main project.

Failure to sync your fork before pushing changes and creating a merge request can result in your merge request being denied until synced properly.

To sync, navigate to your cloned repo in a terminal window and type the following:

```
git fetch upstream
```

Then make sure your on the master branch by typing:

```
git checkout master
```

Then merge the upstream master branch with your master branch by:

```
git merge upstream/master
```

You might have to resolve some merge conflicts at this point. See the Git documation or search Google for issues you might run into.

Note: You can sync your forked project with the main project as often as you like. Syncing often usually reduces potential merge conflicts.

Push Changes

Pushing your changes it simple and easy. Simple type the following in a terminal window from within your cloned repo:

```
git push origin master
```

This pushes the changes you've committed to your forked projects repository.

Create Merge Request

A merge request is how the changes you pushed to your forked project get into the main project. Login to gitlab, go to your forked project, click the merge tab, then "New merge request". Select the master branch in your forked copy and click "Compare branches". You should be taken to the next step where you can give the merge request a title and a description. The description is very important for the approval team to understand what the merge is related to. This page will also show the commits that will be merged and the file diffs. When your ready click "Submit merge request".

Wait for Approval

Approval may be the next day or make take up to a week depending on complexity and schedules. Approvals are done Monday-Friday 8am - 5pm EST.

When your merge request is accepted or denied you will get an email notice regarding its status.

Pull Changes

Pulling in the changes that were merged into the main project can be done through the admin interface for the HUB. You must have admin rights to access the administrator interface.

Testing

Overview

There's a lot of information and articles out there about why and how you should be writing your tests. But, in short, it ensures your code does what you want it to and makes sure it continues to even after you or others make modifications.

Not everything is easy to test. But, at the very least, libraries and other shared pieces of code would ideally be covered by unit tests. It takes time, but it's definitely easier to do it as you write code than it is to go back and add all your tests at the end of the project.

In an effort to make testing a little more accessible, HUBzero offers some basic guidance and structure for your testing. We'll step through what's available below.

Location

Your tests will live in a Tests folder within the applicable extension directory. For example, the database library tests can be found at:

```
core/libraries/Hubzero/Database/Tests/...
```

Test naming convention and structure should follow the definitions found in the [PHPUnit documentation.](#)

Test Types

There are two primary types of tests, basic and database. Basic tests involve no external resources, whereas database tests require the ability to simulate/mock database calls.

Basic Tests

Basic tests in HUBzero offer no additional functionality or abstraction over the PHPUnit_Framework_TestCase. There's therefore really nothing that needs to be covered here that isn't already in the PHPUnit documentation.

Database Tests

To help with writing tests that require a database object, we've worked to provide some shortcuts and best practices. Database tests are tough because they can be slow, and you don't

want them to depend a certain database state or mess up another developers database. So, to get around this, we either completely mock the database, or use a reloadable sqlite database.

Let's look at an example of this.

```
/**
 * Test to make sure we can run a basic select statement
 *
 * @return void
 **/
public function testBasicFetch()
{
    $dbo   = $this->getMockDriver();
    $query = new Query($dbo);

    // Try to actually fetch some rows
    $rows = $query->select('*')
                ->from('users')
                ->whereEquals('id', '1')
                ->fetch();

    // Basically, as long as we don't get false here, we're good
    $this->assertCount(1, $rows, 'Query should have returned one resul
t');
}
```

You'll see above the call to a function named getMockDriver(). This method is going to give you back a database object, loading up a sqlite database named test.sqlite3, by default. This is a fully functioning database driver.

To make the database driver useful, you'll need at least two files included in your tests directory. They are:

```
Tests/Fixtures/seed.xml
Tests/Fixtures/test.sqlite3
```

The seed.xml file will contain all your sample data. This will be automatically loaded in the test framework for each test class (i.e. file). The structure and destination of all database operations will come from the test.sqlite3 file.

The names of those files can also be changed by overwriting the $fixture and/or $seed properties on your test class.

Scaffolding and Running Tests

To get starting writing new tests, you can use the muse scaffolding command to create a test stub. This will look something like this:

```
me@me.org:~# muse scaffolding create test lib_database --type=database
Creating /var/www/example/core/libraries/Hubzero/database/Tests/Exampl
eDatabaseTest.php
```

The test scaffolding expects the first argument after test to be the extension into which the test should be placed. A --type argument can also be given to specify whether or not you're creating a basic or a database test. In this example, given that we're testing the database object, the database test type obviously makes sense.

Running Tests

Once you've created your tests, you'll need to run them. To get started, you can use the muse test command.

```
me@me.org:~# muse test run lib_database
PHPUnit 4.6.2 by Sebastian Bergmann and contributors.

.................................................

Time: 2.51 seconds, Memory: 17.5Mb

OK (51 tests, 73 assertions)
```

More details on the muse functionality can be found in the [Muse documentation](#).

API

Internal Requests via JavaScript

Internal requests to the API can easily be handled through the XMLHttpRequest feature of JavaScript. Many JavaScript libraries abstract away the details of this feature, and jQuery is no different.

In a typical API call, one would be expected to provide authentication in the form of a user access token. This token is often retrieved by asked the user to authenticate to the API server. But, because the client and the server in this scenario are the same, and the user is already logged in, it seems silly to ask them to authenticate again. Even still, we need a token to make an authenticated call to the API. To do this, we must then request a token based on the clients credentials in the form of an HTTP cookie, much the same way standard sessions are managed with a browser.

To make this even simpler, we've added some snippets to the HUBzero libraries to help you out. To setup the API request in your jQuery code, simply include the initApi call, as shown:

```
jQuery(document).ready(function($) {
    Hubzero.initApi(function() {
        // your code here
    });
});
```

All this does is it makes a request to the API to get a token based on your current session cookie. It then sets that token as a header that will be used by jQuery in all future $.ajax requests.

Lastly, you'll need to make sure the Hubzero JavaScript object is available. To do so, just make sure to include the Html environment call somewhere in your PHP to include our core JS library.

```
Html::behavior('core');
```

AWS

Connecting via ssh

On the EC2 instances page, select your instance and click the Connect button. Your ssh connection command will look similar to this, depending on where your Amazon private key file is kept.
"ssh -i private_key.pem root@ec2-52-23-170-128.compute-1.amazonaws.com"

Depending on your user access configuration, "centos" may be used as the username. For example, "ssh centos@your.hostname.com"

Sudo to root

To sudo to root, enter the following command when logged in a the "centos" user:

"sudo su -"

You will change to the root user and placed in the root user's home directory.

The hub configuration file

The hub configuration file is located in the hub document root. Located at "/var/www/hub"

Within that directory you will find a "configuration.php" file which contain many of the configuration options set by the CMS from the global configuration page in the administrator interface (/administrator) of the hub. You will also find the web username and password for the database there.

Connecting to the database as the web user with a database client

Connections to the database are restricted to localhost for security purposes.

In your database client you must select the SSH option and enter the appropriate credentials for the SSH tunnel and Database to connect.

The SSH credentials can be the "centos" user and the stored key on your workstation. Other

Linux users with SSH access may be sued as well.

The database credentials to connect as the web user are found in the hub configuration.php file (var $password and var $db). You may also use another database account that has been previously granted database access.

Creating or granting additional permissions

New Linux users should be created at the system level.

New Database users and granting permission should be done by the root user. As root, you may enter MySQL by typing "mysql" at the command line, without a password.