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:

- (X)HTML
- Cascading Stylesheets (CSS)
- JavaScript (familiarity with the MooTools 1.11 framework is a plus)
- XML
- Model-View-Controller (MVC) design pattern
- Object-Oriented Programming

Installation

Directories & File Structure

The initial directory structure of a HUBzero install.

```
/hubzero
  /administrator
  /cache
  /components
  /images
  /includes
  /language
  /libraries
  /logs
  /media
  /modules
  /plugins
  /site
  /templates
  /tmp
  /xmlrpc
  configuration.php
  index.php
  index2.php
  htaccess.txt
  robots.txt
```

While this looks very much like a typical Joomla! 1.5 install, there are some noticeable exceptions. Some directories vital to HUBzero functionality have been added. A quick explanation of the additional directories:

/site
: This is where HUB specific data such as member pictures, files used in wiki pages, etc. is stored.

Accessing Files

Accessing via SSH

The following tutorial should help you in using SSH to connect to and from your HUBzero server(s). You should be relatively comfortable with using a terminal (also referred to as a "command-line tool") to navigate directories and manipulate files.

Warning: Most accounts do **not** have SSH/sFTP access initially. Your system administrator must grant your account access before you will be able to connect.

From a terminal type ssh <user>@<host>. You will then be prompted for a password. Both the username and password will typically be the same as the account you registered on <host>.

```
yourmachine:~ you$ ssh username@host
yourmachine:~ you$ username@host password:

host ~
```

Windows Clients

- PuTTY (a Telnet and SSH client)

Mac OSX

All versions of Mac OSX come with Terminal.app which may be found in the /Utilities directory of your /Applications directory.

Accessing via sFTP

sFTP, or secure FTP, is a program that uses SSH to transfer files. Unlike standard FTP, it encrypts both commands and data, preventing passwords and sensitive information from being transmitted in the clear over the network. It is functionally similar to FTP, but because it uses a different protocol, you can't use a standard FTP client to talk to an sFTP server, nor can you connect to an FTP server with a client that supports only sFTP.

The following tutorial should help you in using sFTP to connect to and from your HUBzero server(s).

Warning: Most accounts do **not** have SSH/sFTP access initially. Your system administrator

must grant your account access before you will be able to connect.

Graphical Clients

Using graphical SFTP clients simplifies file transfers by allowing you to transmit files simply by dragging and dropping icons between windows. When you open the program, you will have to enter the name of the host (e.g., yourhub.org) and your HUB username and password.

Windows Clients

- WinSCP
- BitKinex
- FileZilla
- PuTTY

Mac OSX Clients

- Transmit
- Fetch
- Cyberduck
- Flow
- Fugu

Command-line

You can use command line SFTP from your Unix account, or from your Mac OS X or Unix workstation. To start an SFTP session, at the command prompt, enter:

```
yourmachine:~ you$ sftp username@host
yourmachine:~ you$ username@host password:
```

host ~

Some standard commands for command-line sFTP

Command	Description
cd	Char
chmod	Char comp
chown	Char comp
dir (or ls)	List t remo
exit (or quit)	Close and e
get	Copy local

Command	Description
help (or ?)	Get h
lcd	Char
lls	See a the lo
lmkdir	Crea
ln (or symlink)	Crea comp
lpwd	Show direc
lumask	Char
mkdir	Crea
put	Copy remo
pwd	Show direc
rename	Rena
rm	Delet
rmdir	Rem direc
version	Displ
!	In Ur enter SFTP !pwd) dropp

Finding Files

Once connected to a server, by either sFTP or directly with SSH, you will need to find the web root which contains the HUB install. The web root for the production version of a HUB can be found at /www/yourhub. Typically, HUBs will also have a development version of a HUB, which can be found at /www/dev.

Once in the desired directory, file layout and directory structure follows Joomla! 1.5 conventions unless otherwise noted.

See the Installation overview for details on a typical HUBzero install's directory structure.

Direct Database Access

Accessing via command-line

The following tutorial should help you in using SSH to connect to and from your HUBzero server(s) and access the database. You should be relatively comfortable with using a terminal (also referred to as a "command-line tool") to navigate directories and manipulate files.

Warning: Most accounts do **not** have SSH/sFTP access initially. Your system administrator must grant your account access before you will be able to connect.

See Accessing Files for further details on how to use SSH.

Libraries

Hubzero

Location:

```
/libraries/Hubzero
```

The Hubzero library contains code that is essential for a hub to run properly and altering or adding to the library without Hubzero approval is *strongly* discouraged.

File Formatting

For files that contain only PHP code, the closing tag ("?>") is omitted. It is not required by PHP, and omitting it prevents the accidental injection of trailing white space into the response.

Class Names

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 "/libraries/Hubzero/User/Helper.php" must map to the class name "Hubzero_User_Helper".

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 "Hubzero_PDF" is not allowed while "Hubzero_Pdf" is acceptable.

Note: Code deployed alongside Hubzero libraries must never start with "Hubzero_".

Filenames

Hubzero standardizes on a class naming convention whereby the names of the classes directly map to the directories in which they are stored. The root level directory of Hubzero's standard library is the "/libraries/Hubzero" directory. All Hubzero classes are stored hierarchically under this root directory.

For all other files, only alphanumeric characters, underscores, and the dash character ("-") are permitted. Spaces are strictly prohibited.

File names must map to class names as described above.

Debugging

Joomla's Debugging Mode

To turn on Joomla!'s Debug mode:

- Login to the Joomla administration 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.

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" />
```

Coding Styles and Conventions

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 standardizes on a 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 "Hubzero_User_Profile".

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 "Hubzero_PDF" is not allowed while "Hubzero_Pdf" 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_".

Joomla! 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). One exception is for Joomla framework classes which must begin with an uppercase 'J' with the next letter also being uppercase.

```
JHtmlHelper
JXmlParser
JModel
```

These conventions define a pseudo-namespace mechanism for the Joomla framework and HUBzero core library. Third-party developers are to avoid beginning names with an uppercase 'J' or 'Hubzero' as they are 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 JController
{
    // Methods
}
```

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

```
class BannerControllerClient extends JController
{
    // Methods
}
```

Models

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

```
class BannerModelClient extends JModel
{
    // Methods
}
```

Plugins

The naming convention is plg[Folder][Element]

```
class plgContentPagebreak extends JPlugin
{
    // 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/Factory.php

Hubzero/Session/Helper.php

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

Hubzero library file names must map to class names as described above. Joomla extension names generally follow similar conventions and will map to class names as described below.

Controllers

For single controller components, the naming convention of [Component]Controller will map to a file name of controller.php and be located in the component folder.

```
com_content
 /controller.php
```

For a multi-controller components, such as the Banners in the Administrator, the convention of [Component]Controller[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_banner
  /controllers
    /banner.php
    /client.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_banner
  /models
    /banner.php
```

```
    /client.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 “default”.

```
com_content
  /views/
    /article/
      /tmpl/
        / default.php
        / default.xml
        / form.php
        / form.xml
        / pagebreak.php
        / pagebreak.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 are permitted only in instance variables. 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. This is the only acceptable application of an underscore in a variable name. 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.

```
class Hubzero_Example
```

```
{
    private $_status = null;

    protected $_fieldName = null;

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

Constants

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.

Common Tasks & Objects

Config

Joomla! Configuration

Accessing the global Joomla! site configuration:

```
$jconfig =& JFactory::getConfig();
```

Retrieving a value from the configuration:

```
echo $config->getValue('config.sitename');
```

HUB Configuration

Although rarer than accessing the global Joomla! site configuration, sometimes it is necessary to access HUB-specific configurations. This can be done as follows:

```
$xhub =& XFactory::getHub();
```

Retrieving a value from the configuration:

```
echo $xhub->getCfg('hubShortName');
```

Users & Profiles

Joomla User Object

Current User

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

```
$juser =& JFactory::getUser();
```

Other Users

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

```
$otheruser =& JUser::getInstance($id);
```

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

```
$id = $juser->get('id');
$name = $juser->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. Vint Cerf)
- username - The login/screen name of the user. (e.g. shmuffin1979)
- 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 Joomla!. (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' in Joomla!.
- 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.

```
$user =& JFactory::getUser();
$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 XProfile object and load() method (accepts user ID number or username).

```
// Import the needed library
ximport('Hubzero_User_Profile');

// Instantiate a new profile object
$profile = new Hubzero_User_Profile();

// Load the profile
$profile->load( $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 as follows:

```
// Call the user object
$juser =& JFactory::getUser();

// If 'guest' is true, they are logged OUT
// If 'guest' is false, they are logged IN
if ($juser->get('guest')) {
 return false;
}
```

Privileges

Not all authenticated users are given equal rights. For instance, a Super Administrator may be able to edit anyone's content, while a Publisher may only be able to edit their own. The authorize() member function can be used to determine if the current user has permission to do a certain task. The first parameter is used to identify which component or function we wish to authenticate against. The second represents the task. The third and fourth are optional; they further break the permissions down into record types and ownership respectively.

In Joomla! 1.5, the rights for all of the core components are stored in libraries/joomla/user/authorization.php. These are available to all extensions wherever authentication is required. If the permission scheme of the Content component suits your extension's needs, you can use code similar to the following to determine what functions to give to a specific user.

```
$user =& JFactory::getUser();

if ($user->authorize('com_content', 'edit', 'content', 'all')) {
 echo "<p>You may edit all content.</p>";
} else {
 echo "<p>You may not edit all content.</p>";
}

if ($user->authorize('com_content', 'publish', 'content', 'own')) {
 echo "<p>You may publish your own content.</p>";
} else {
 echo "<p>You may not publish your own content.</p>";
}
```

The permissions for core functions may not be suitable for your extension. If this is the case, you can create your own permissions. You will probably want to add this code in a place where it will always be executed, such as the beginning of the component you are building or in a systemwide plugin. First, you need to get an authorization object using the getACL() member function of JFactory. This works like getUser() in that it only creates one authorization object during any particular Joomla! request. Once you have this object, call the addACL() member function to add permissions. Pass in the name of your component or function, the task name, the string 'users', and the user type (in lowercase) respectively. If you want to also define record sets and ownership, pass those in as an additional two parameters.

Note that in Joomla! 1.5, permissions are not inherited. For example, if you give an Administrator the right to edit content, Super Administrators do not automatically get this right; you must grant it separately.

```
$auth =& JFactory::getACL();

$auth->addACL('com_userinfo15', 'persuade', 'users', 'super administra
tor');
$auth->addACL('com_userinfo15', 'persuade', 'users', 'administrator');
$auth->addACL('com_userinfo15', 'persuade', 'users', 'manager');

$user =& JFactory::getUser();

if ($user->authorize('com_userinfo15', 'persuade')) {
 echo "<p>You may persuade the system to do what you wish.</p>";
} else {
 echo "<p>You are not very persuasive.</p>";
}
```

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 user object
$juser =& JFactory::getUser();

// Include a needed HUBzero library
ximport('Hubzero_User_Helper');

// Get the groups of the current logged-in user
$user_groups = Hubzero_User_Helper::getGroups( $juser->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

Database

Overview

Joomla! 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 Joomla! database layer ensures a maximum of compatibility and flexibility for your extension.

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

Preparing The Query

```
// Get a database object
$db =& JFactory::getDBO();

$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 =& JFactory::getDBO();
$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 Joomla! it is most often used for updating or administering the database simple because the various load methods detail on this page have the query step built in to them.

The syntax is very straightforward:

```
$db =& JFactory::getDBO();
$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 =& JFactory::getDBO();
$query = "
  SELECT COUNT(*)
    FROM ".$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 =& JFactory::getDBO();
$query = "
  SELECT ".$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

loadResultArray()

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

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

$db->setQuery($query);
$column= $db->loadResultArray();
print_r($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. loadResultArray() is equivalent to loadResultArray(0)

loadResultArray($index)

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

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

$db->setQuery($query);
$column= $db->loadResultArray(1);
print_r($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']
```

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

```
$db->setQuery($query);
for ( $i = 0; $i loadResultArray($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
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.

JTable

Overview

The JTable class is an implementation of the Active Record design pattern. It is used throughout Joomla! for creating, reading, updating, and deleting records in the database table.

When properly extended, JTable gives you all of the basic functions you need for managing and retrieving records in a database table. Member functions take care of the rest when you add member variables, the table name, and the key column.

Writing an extension of JTable

To use JTable, create an extension of the class. In this example, we have a database table containing recipes.

```
<?php

defined('_JEXEC') or die();

class TableRecipes extends JTable
{
 var $id = null;
 var $ingredients = null;
 var $instructions = null;
 var $serves = null;
 var $difficulty = null;
 var $prep_time = null;
 var $cook_time = null;
 var $published = 0;

 function __construct(&$db)
 {
  parent::__construct( '#__recipes', 'id', $db );
 }
}
```

When naming your class extension, the convention is to prefix it with 'Table', then follow with a CamelCased version of the table's name. All of the member variables of your class should match the column names in the database. The default values should be valid according to the table schema. For instance, if you have columns that are NOT NULL, you must use a value

other than 'null' as the default.

Finally, create a constructor for the class that accepts a reference to the current database instance. This will call the parent constructor which needs the name of the table, the name of the primary key column, and the database instance. The name of the table uses #__ instead of jos_, as the administrator can pick any table prefix desired during Joomla! installation.

If you were using this class as a part of a component called 'Recipes', you would place this code in the file /administrator/components/com_recipes/tables/recipes.php.

Using a JTable class extension

Once the table class is in place, you can use it in any Joomla! extension. To include the file, place this line in your extension's source code (use com_nameofyourcomponent in place of com_recipes):

```
JTable::addIncludePath(JPATH_ADMINISTRATOR.DS.'components'.DS.'com_recipes'.DS.'tables');
```

To get an instance of the object, use this code:

```
$row =& JTable::getInstance('recipes', 'Table');
```

Notice that the lowercase version of the suffix of your class name is used as the first parameter, with the prefix 'Table' as the second. Also, the getInstance() member function of JTable returns the object by reference instead of value.

In a model class (extends JModel) you can also use:

```
$row =& $this->getTable('recipes');
```

Notice that if you have not used the standard naming convention, you can supply the class prefix as the optional second parameter.

Create/Update

In a typical situation, you will have an HTML form submitted by the user which PHP will interpret for you as an associative array. The JRequest class in Joomla! has functions ready to assist with retrieving this data safely. Use JRequest::get('post') to retrieve all of the elements in the HTTP POST request as a sanitized array.

Once you have this array, you can pass it into the bind() method of JTable. Doing this will match the associated items of the array with member variables of the class. In the following example, the array is retrieved from JRequest::get('post') and immediately passed into bind().

```
if (!$row->bind( JRequest::get( 'post' ) )) {
 return JError::raiseWarning( 500, $row->getError() );
}
```

If bind() fails, you want to stop the application and explain the failure before your extension attempts to send the data. The raiseWarning() function of JError allows you to stop Joomla!, while the getError() function returns the error message stored in the JTable object.

When binding succeeds and your object is ready, call the store() function. Again, if something goes wrong, stop the application and explain why.

```
if (!$row->store()) {
 JError::raiseError(500, $row->getError() );
}
```

Note:

- If any member variables of your JTable object are null when store() is called, they are ignored by default. This allows you to update specific columns of your table, while leaving the others untouched. If you wish to override this behavior to ensure that all columns have a value, pass true into store().
- The JTable::bind() and JRequest::get() functions do not enforce data types. If you need a column to be a specific type (for instance, integer), you need to add this logic to your code before calling store().

Read

To load a specific row of the database with JTable, pass the key into the load() member function.

```
$row->load( $id );
```

This relies on the key column you specified in the second parameter of parent::__construct() when you extended JTable.

Delete

Like read(), delete() allows you to destroy a specific row in the table based on the key specified earlier.

```
$row->delete( $id );
```

If you want to delete multiple rows at once, you will need to write the query manually.

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

require_once( JPATH_ROOT.DS.'components'.DS.'com_tags'.DS.'helpers'.DS
.'handler.php' );

class ExampleTags extends TagsHandler
{
```

```
 public function __construct( $db )
 {
  // The database connection object
  $this->_db  = $db;
  // A unique name
  $this->_tbl = 'example';
 }
}
```

When naming your class extension, the convention is to have a CamelCased version of the component's name suffixed with "Tags".

Finally, create a constructor for the class that accepts a reference to the current database instance and 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 = trim(JRequest::getVar( 'tags', '' ));

// Get the database object
$database =& JFactory::getDBO();

// Instantiate the tagging class
$et = new ExamplesTags( $database );

// Tag the object
$et->tag_object( $juser->get('id'), $object_id, $tags );
```

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

get_tag_cloud($showsizes, $showadmintags, $object_id)

Returns a string of comma-separated tags.

```
// Get the database object
$database =& JFactory::getDBO();

// Instantiate the tagging class
$et = new ExamplesTags( $database );

// Get a tag cloud (HTML List)
$tags = $et->get_tag_cloud( $showsizes, $showadmintags, $object_i
d );
print_r($tags);
```

will give:

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

get_tag_cloud($showsizes, $showadmintags, $object_id)

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.

```
// Get the database object
$database =& JFactory::getDBO();

// Instantiate the tagging class
$et = new ExamplesTags( $database );

// Get a string of tags separated by commas
$tags = $et->get_tag_string( $object_id );
print_r($tags);
```

will give:

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

get_tags_on_object($object_id)

Returns an array of associative arrays.

```
// Get the database object
$database =& JFactory::getDBO();

// Instantiate the tagging class
$et = new ExamplesTags( $database );

// Get a string of tags separated by commas
$tags = $et->get_tags_on_object( $object_id );
print_r($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:

```
// Load the plugin
JPluginHelper::importPlugin( 'hubzero' );
$dispatcher =& JDispatcher::getInstance();

// Trigger the event
$tf = $dispatcher->trigger( 'onGetMultiEntry', array(array('tags','tag
s','actags','',$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.

Search

Retrieving GET & POST data

JRequest 'getVar' method

To retrieve GET/POST request data, Joomla! uses the getVar method of the JRequest class (JRequest::getVar()).

Retrieving Data

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

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

Unless other parameters are set, all HTML and trailing whitespace will be filtered out.

The DEFAULT Parameter

If you want to specify a default value in the event that 'address' is not in the request or is unset, use this code:

```
$address = JRequest::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 = JRequest::getVar('address', 'default value goes here', 'pos
t');
```

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 = JRequest::getVar('address', 'default value goes here', 'pos
t','variable type');
```

Here is a list of types you can validate:

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

The FILTER MASK Parameter

Finally, there are some mask constants you can pass in as the fifth parameter that allow you to bypass portions of the filtering:

```
$address = JRequest::getVar('address', 'default value goes here', 'pos
t','validation type','mask type');
```

- JREQUEST_NOTRIM - prevents trimming of whitespace
- JREQUEST_ALLOWRAW - bypasses filtering
- JREQUEST_ALLOWHTML - allows most HTML. If this is not passed in, HTML is stripped out by default.

Constants

Joomla! Constants

These constants are defined for use in Joomla and extensions:

DS	Directory separator. "/"
JPATH_ADMINISTRATOR	The path to the administrator folder.
JPATH_BASE	The path to the installed Joomla! site.
JPATH_CACHE	The path to the cache folder.
JPATH_COMPONENT	The path to the current component being executed.
JPATH_CONFIGURATION	The path to folder containing the configuration.php file.
JPATH_INSTALLATION	The path to the installation folder.
JPATH_LIBRARIES	The path to the libraries folder.
JPATH_PLUGINS	The path to the plugins folder.
JPATH_ROOT	The path to the installed Joomla! site.
JPATH_SITE	The path to the installed Joomla! site.
JPATH_THEMES	The path to the templates folder.
JPATH_XMLRPC	The path to the XML-RPC Web service folder.

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

For URL paths, try using JURI::base.

Extensions (general)

Overview

Joomla! already is a rich featured content management system but if you're building a website with Joomla! and you need extra features which aren't available in Joomla! by default, you can easily extend it with extensions. There are five types of extensions for Joomla!: 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. 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 perfectly sense as a stand-alone system. A component will be shown in the main part of your website and only one component will be shown. A menu is then in fact nothing more then a switch between different components.

Hubzero Components

- com_answers
- com_blog
- com_citations
- com_contribtool
- com_contribute
- com_documentation
- com_events
- com_features
- com_feedback
- com_groups
- com_hub
- com_infrastructure
- com_jobs
- com_kb
- com_meetings
- com_members
- com_myhub
- com_register
- com_resources
- com_sef
- com_store

- com_support
- com_tags
- com_tools
- com_topics (alternate name for com_wiki)
- com_usage
- com_whatsnew
- com_wiki (alternate name for com_topics)
- com_wishlist
- com_xflash
- com_ximport
- com_xpoll (supplants Joomla's com_poll)
- com_xsearch (supplants Joomla's com_search)

Modules

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

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

Hubzero Modules (front-end)

- mod_events_cal
- mod_events_latest
- mod_featuredmember
- mod_featuredquestion
- mod_featuredresource
- mod_feed_youtube
- mod_findresources
- mod_mycontributions
- mod_myfavorites
- mod_mygroups
- mod_mymeetings

- mod_mymessages
- mod_mypoints
- mod_myprofile
- mod_myquestions
- mod_mysessions
- mod_mysubmissions
- mod_mytickets
- mod_mytools
- mod_mywishes
- mod_notices
- mod_polltitle
- mod_popularfaq
- mod_popularquestions
- mod_quicktips
- mod_quotes
- mod_randomquote
- mod_rapid_contact
- mod_recentquestions
- mod_reportproblems
- mod_resourcemenu
- mod_slideshow
- mod_sliding_panes
- mod_spotlight
- mod_toptags
- mod_twitterfeed
- mod_whatsnew
- mod_wishvoters
- mod_xflash
- mod_xlogin
- mod_xlogin_mini
- mod_xpoll
- mod_xsearch
- mod_xwhosonline

Hubzero Modules (back-end/administrative)

- mod_dashboard

Plugins

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

Hubzero Plugins

- content
    - xhubtags
- groups
    - forum
    - members
    - messages
    - resources
    - wiki
    - wishlist
- members
    - blog
    - contributions
    - favorites
    - groups
    - messages
    - points
    - resources
    - resume
    - topics
    - usage
- resources
    - citations
    - favorites
    - questions
    - recommendations
    - related
    - reviews
    - share
    - supportingdocs
    - usage
    - versions
    - wishlist
- support
    - xfeed
    - xhub
- system
    - xfeed
    - xhub
- tageditor
    - autocompleter
- tags
    - answers
    - blogs
    - events

        - groups
        - members
        - resources
        - support
        - topics
- usage
        - chart
        - domainclass
        - domains
        - maps
        - overview
        - partners
        - region
        - tools
- whatsnew
        - content
        - events
        - kb
        - resources
        - topics
- xauthentication
        - hzldap
- xhub
        - xlibrary
- xmessage
        - email
        - handler
        - im
        - internal
        - rss
        - smstxt
- xsearch
        - answers
        - blogs
        - content
        - events
        - kb
        - members
        - resources
        - tags
        - topics

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.

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 Joomla! 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 Joomla! installation and check the components menu, the module manager and the plugin manager. Joomla! comes with a number of core components, modules and plugins. By checking what theyâ€™re doing, the difference between the three types of building blocks should become clear. You can also check out the official Joomla! extensions page. Browse through the extension categories and youâ€™ll be amazed about the extension possibilities you have for your site.

Installing

Installing From Package

Warning: Unlike a typical Joomla! 1.5 install, most HUBs do **not** have public write access to the various extensions directories. Using this method may fail as a result. Contact your system administrator for any necessary changes.

Joomla! 1.5 provides a convenient Installer utility in the administrative back-end. From here, one can install new modules that have been packaged as .zip files. The installer moves all the necessary files to their appropriate locations and creates any database entries needed.

Note: There is usually an upper limit to the size of files that can be uploaded within the web server itself. This limit is set in the PHP configuration file and may differ between web servers and hosts. Current HUB installs set the limit to **100MB**. This limit cannot be altered from within Joomla!. Contact your system administrator for help if needed.

1. Log in to the administrative back-end of the HUB you wish to install the module on.
2. Once logged-in navigate to the Extensions Installer. This can be found from the main menu by following the "Install/Uninstall" option found in the drop-down under "Extensions".
3. Under "Upload Package File", click on the "Browse" (note: some browsers/OSes may have alternate wording) button. This will open the File Upload dialogue window. Navigate to the location of the desired package file on the local hard drive. Select the extension file and click the Open button. The dialogue window will disappear and the path to, and name of, the extension file will appear in the File Upload field.
4. Click the "Upload File & Install" button to complete the transfer and installation of a copy of the extension files from the local computer to the /yourhub/{ExtensionType}/ directory tree. Note: Any language files packaged with the module will be moved to their respective sub-directories of the /yourhub/language/ directory.

Installing From Directory

Joomla! 1.5 provides a convenient Installer utility in the administrative back-end. From here, one can install new modules from an existing directory on the server. The installer moves all the necessary files to their appropriate locations and creates any database entries needed.

1. If the module is packaged as a .zip file, unpack it onto the local hard drive before uploading.
2. Upload the entire contents of the module via SSH/sFTP. Ideally the file should be transferred to the /www/yourhub/administrator/components/com_installer/module/yourmodulename directory of Joomla!.

See Accessing Files for further details on how to use SSH/sFTP.

3. Log in to the administrative back-end of the HUB.
4. Once logged-in navigate to the Extensions Installer. This can be found from the main menu by following the "Install/Uninstall" option found in the drop-down under "Extensions".
5. Under "Install from Directory" enter the exact location of the module file (it must be the **absolute** location) in this example: /www/yourhub/administrator/components/com_installer/module/yourmodulename.
6. Click the "Install" button to complete the installation. The appropriate module files will be moved to the /yourhub/modules/ directory tree. Note: Any language files packaged with the module will be moved to their respective sub-directories of the /yourhub/language/ directory.

Installing By Hand

Installing an extension by hand requires a few more steps than the Joomla! Extensions Installer but is still a fairly easy and quick process.

1. If the extension is packaged as a .zip file, extract the files to a location on your local machine.

2. Upload the entire contents of the extension, except language files, via SSH/sFTP to the /yourhub/{ExtensionType}/ directory. Any language files associated with the extension must be copied to their respective sub-directories of the /yourhub/language/ directory.

   Components are unique in that they will typically have files installed in two locations: /components and /administrator/components.

Extension Type	Install Location
Component	/yourhub/components/{ExtensionName} /yourhub/administrator/components/{ExtensionName}
Module	/yourhub/modules/{ExtensionName}
Plugin	/yourhub/plugins/{PluginType}/
Template	/yourhub/templates/{ExtensionName}

   See Accessing Files for further details on how to use SSH/sFTP.

3. Log in to the administrative back-end of the HUB.

4. **Components**

1. Components do not technically need a database entry to function in their simplest form. However, an entry is needed if one wishes to use parameters or have the component appear under the "Components" list in the administrative back-end. This must be done by hand via MySQL command-line, some form of MySQL database GUI, or executing a PHP script. A sample SQL is provided below:

```
INSERT INTO #__components(
 `id`,
 `name`,
 `link`,
 `menuid`,
 `parent`,
 `admin_menu_link`,
 `admin_menu_alt`,
 `option`,
 `ordering`,
 `admin_menu_img`,
 `iscore`,
 `params`,
 `enabled`
)
VALUES(
 '',
 'My Component',
 '',
 0,
 0,
 'option=com_mycomponent',
 'My Component',
 'com_mycomponent',
 0,
 'js/ThemeOffice/component.png',
 0,
 '',
 1
);
```

   See Direct Database Access for further details on how to access a HUB's database via command-line or GUI utility.

Modules

1. Once logged-in navigate to the Modules Manager. This can be found from the main menu by following the "Modules Manager" option found in the drop-down under "Extensions".
2. Click the "New" button in the toolbar. This will present you with a list of all available modules, including those with existing directories but no database entries (such as the one you just copied to /yourhub/modules/).
3. Find the name of your newly added module and click its radio button. Once selected, click the "Next" button in the toolbar. This will take you to an "edit module" screen where you may enter a title, adjust parameters, select a position, etc.
4. Enter a title, adjust parameters, select a position, and enter any other necessary information. Click "Save" in the toolbar.

Plugins

1. Unlike modules, there is no convenient Joomla! utility to create the necessary database entry for us. This must be done by hand via MySQL command-line, some form of MySQL database GUI, or executing a PHP script. A sample SQL is provided below:

```
INSERT INTO #__plugins(
 `id`,
 `name`,
 `element`,
 `folder`,
 `access`,
 `ordering`,
 `published`,
 `iscore`,
 `client_id`,
 `checked_out`,
 `checked_out_time`,
 `params`
)
VALUES(
 '',
 'System - Hello World',
 'helloworld',
 'system',
 0,
 1,
 1,
 0,
 0,
 0,
 '0000-00-00 00:00:00',
 ''
```

```
);
```

See Direct Database Access for further details on how to access a HUB's database via command-line or GUI utility.

Templates

1. Once logged-in navigate to the Templates Manager. This can be found from the main menu by following the "Template Manager" option found in the drop-down under "Extensions".
2. Here you will be presented with a list of available templates. Your newly added template should be available. To make it the default template of the site, select it by clicking the radio button next to its name.
3. Click the "Default" button to make the template the default.

Uninstalling

Overview

If you wish to uninstall an extension on your Joomla! site, then follow these simple steps:

1. Select "Extensions" and then "Install / Uninstall" from the drop-down menu
2. Select the type of extension you wish to uninstall. You will have the choice between Components, Modules, Plugins, Languages and Templates.
3. Find the extension you wish to uninstall and check the checkbox to the left of the extension title.
4. In the upper-right corner of the screen, press "Uninstall"

It's as simple as that. If Joomla! can't uninstall the extension, you will be prompted with an error message. If this happens, it's most likely to be caused by the extension. As extensions are developed by third-party volunteers, you will have to try to get support from the developers of the specific extension.

Parameters

Overview

Coming soon.

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 extensionsExtension 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 JText::_('MYLINE'); ?></strong>
```

```
<!-- mycomponent layout -->
<div>
 <!-- Load the module -->
 <php echo XModuleHelper::renderModule('mymodule'); ?>
 <!-- Translate some component text -->
 <php echo JText::_('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 JText::_('MOD_MYMODULE_MYLINE'); ?></strong>
```

```
<!-- mycomponent view -->
<div>
 <!-- Load the module -->
 <php echo $this->Widgets()->renderWidget('mywidget'); ?>
 <!-- Translate some module text -->
 <php echo JText::_('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 (JText) 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 JText::_("MOD_EXAMPLE_MY_LINE"); ?></p>
```

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

Further Help

For further help with language files, including creating and distributing your own translations to existing extensions, see Joomla!'s documentation.

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 perfectly sense as a stand-alone system. A component will be shown in the main part of your website and only one component will be shown. A menu is then in fact nothing more then a switch between different components.

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}.

We also strongly encourage developers to rake a look at Joomla!'s documentation.

Directory Structures & 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 #__components table will not appear in the administrative list of available components.

To illustrate the typical component directory structures and files:

```
/hubzero
  /administrator
    /components
      /com_example
```

```
...
/components
  /com_example
      /models
        foo.php
      /views
        /index
          /tmpl
            default.php
            default.xml
    controller.php
    example.php
    router.php
```

In the above example, all component related files and sub-directories are split between the administrator components and front-end components. In both cases, the files are contained within directories titled "com_example". Some directories and files are optional but, for this example, we've included the most common setup.

The file structure in the administrative portion of the component is exactly the same as in the front side. Note that the view, models, controllers etc. of the front and admin parts are completely separated, and have nothing to do with each other - the front part and the admin part can be thought of as two different components! A view in the /administrator/components/com_example folder may have a counterpart with the same name in the /components/com_example folder, yet the two views have nothing in common but their name.

Directory & File Explanation

/com_{componentname}/{componentname}.php
This is the component's main file and entry point *for the front-end part*.
/com_{componentname}/controller.php
This file holds the default frontend controller, which is a class called "{ComponentName}Controller". This class must extend the base class "JController".
/com_{componentname}/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}/view.html.php
This file is the entry point for the view {ViewName}. It should declare the class

{ComponentName}View{ViewName}. This class must extend the base class "JView".

/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 "JModel". 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}/controllers

This folder holds additional controllers, if needed by the application.

/com_{componentname}/controllers/{controllername}.php
This file holds the controller class {ComponentName}Controller{ControllerName}. This class must extend the base class "JController".

Naming Conventions

Classes

The model, view and controller files use the jimport function to load classes from the Joomla! framework, JModel, JView and JController, respectively. Each class is then extended with a new class specific to the component.

The base controller class for the site is named {ComponentName}Controller. For the administrative section, an "s" is added to the ComponentName, giving {ComponentName}sController. Classnames for additional controllers found within the controllers/ subdirectory are {ComponentName}Controller{ControllerName} for site/ and {ComponentName}sController{ControllerName} for admin/.

The view class is named {ComponentName}View{ViewName}.

The model class is named {ComponentName}Model{ModelName}. Remember that the

{ModelName} and the {ViewName} should be the same.

Reserver Words

There are reserved words, which can't be used in names of classes and components.

An example is word "view" (in any case) for view class (except "view" that must be second part of that class name). Because first part of view class name is the same as controller class name, controller class name also can't contain word "view". And because of convension (although violating of it won't produce an error) controller class name must contain component name, so component name also can't contain word "view". So components can't be named "com_reviews", or if thay are, thay must violate naming convention and have different base controller class name (or have some other hacks).

Examples

Here we have a basic front-end component that simply displays a "Hello, World!" message. We present it using both standard Joomla! 1.5 methods and HUBzero methods, which differ in a few key ways. Note, however, that despite any differences from standard Joomla! methods, all HUBzero methods will still work on a stock Joomla! 1.5 install.

Download: Hello World component (Joomla! method)

Download: Hello World component (HUBzero method)

Installation

Installing

See Installing Extensions for details.

Uninstalling

See Uninstalling Extensions for details.

Manifests

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 Joomla! 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_.

Structure

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

A typical component manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<install 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>
</install>
```

Entry Point

Overview

Joomla! 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:

```
index.php?option=com_hello&view=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

Joomla! 1.5 Methodology

The code for this file is fairly typical across components.

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

// Require the base controller
require_once( JPATH_COMPONENT.DS.'controller.php' );

// Require specific controller if requested
if ($controller = JRequest::getWord('controller')) {
 $path = JPATH_COMPONENT.DS.'controllers'.DS.$controller.'.php';
 if (file_exists($path)) {
  require_once $path;
 } else {
  $controller = '';
 }
}

// Create the controller
$classname = 'HelloController'.$controller;
$controller = new $classname( );

// Perform the Request task
```

```
$controller->execute( JRequest::getWord( 'task' ) );

// Redirect if set by the controller
$controller->redirect();
```

The first statement is a security check.

JPATH_COMPONENT is the absolute path to the current component, in our case components/com_hello. If you specifically need either the Site component or the Administrator component, you can use JPATH_COMPONENT_SITE or JPATH_COMPONENT_ADMINISTRATOR.

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.

After loading the base controller, we check if a specific controller is needed. In this component, the base controller is the only controller, but we will leave this conditional check "in place" for future use.

JRequest:getWord() finds a word variable in the URL or the POST data. So if our URL is index.php?option=com_hello&controller=controller_name, then we can retrieve our controller name in our component using: echo JRequest::getWord('controller');

Now we have our base controller 'HelloController' in com_hello/controller.php, and, if needed, additional controllers like 'HelloControllerController1' in com_hello/controllers/controller1.php. Using this standard naming scheme will make things easy later on: '{Componentname}{Controller}{Controllername}'

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 controller might decide to redirect the page, usually after a task like 'save' has been completed. This last statement takes care of the actual redirection.

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 since Joomla! 1.5,

and will be used for all php-only files.

HUBzero Methodology

HUBzero components differ in subtle, but key ways from standard Joomla! components. This is, in part, due to legacy issues. Some changes are made to aid in development while others may simply be a difference in philosophy. Note, however, that **no** differences require hacking or altering Joomla! in any way and HUBzero methodologies will run on any stock Joomla! install.

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

// Check if debugging is turned on
// If it is, we'll turn on PHP error reporting so we can more clearly
see our PHP bugs
$config = JFactory::getConfig();
if ($config->getValue('config.debug')) {
 error_reporting(E_ALL);
 @ini_set('display_errors','1');
}

// Include the JView class
jimport('joomla.application.component.view');

// Require the base controller
require_once( JPATH_COMPONENT.DS.'controller.php' );

// Require specific controller if requested
if ($controller = JRequest::getWord('controller')) {
 $path = JPATH_COMPONENT.DS.'controllers'.DS.$controller.'.php';
 if (file_exists($path)) {
  require_once $path;
 } else {
  $controller = '';
 }
}

// Create the controller
$classname = 'HelloController'.$controller;
$controller = new $classname( );

// Perform the Request task
$controller->execute();

// Redirect if set by the controller
```

```
$controller->redirect();
```

Here, you can see we've added a few things and made one subtle change in calling the execute() method. First, we added some lines that check if site debugging is turned on. If so, we turn on PHP error reporting. This can aid greatly in development.

Next, we added the jimport call to include the component JView class. This is done specifically because HUBzero controllers do **not** extend JController. JController does some autoloading and initial setup for Joomla! components and since we're not employing it, we need to do some class loading and setup of our own.

Finally, we removed the JRequest::getWord('task') being passed to the execute() method. HUBzero controllers handle the task request within the execute() method, rather than passing the task to it.

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.

Most components have two controllers: one for the front-end and one for the back-end.

Creating the Front-end Controller

Joomla! 1.5 Method

Our component only has one task - greet the world. Therefore, the controller will be very simple. No data manipulation is required. All that needs to be done is the appropriate view loaded. We will have only one method in our controller: display(). Most of the required functionality is built into the JController class, so all that we need to do is invoke the JController::display() method.

The code for the base controller com_hello/controller.php is:

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

jimport('joomla.application.component.controller');

/**
 * Hello World Component Controller
 *
 * @package    Joomla.Tutorials
 * @subpackage Components
 */
class HelloController extends JController
{
    /**
     * Method to display the view
     *
     * @access    public
     */
    public function display()
    {
```

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

The JController constructor will always register a display() task and unless otherwise specified (using the registerDefaultTask() method), it will set it as the default task.

This barebones display() method isn't really even necessary since all it does is invoke the parent constructor. However, it is a good visual clue to indicate what is happening in the controller.

The JController::display() method will determine the name of the view and layout from the request and load that view and set the layout. When you create a menu item for your component, the menu manager will allow the administrator to select the view that they would like the menu link to display and to specify the layout. A view usually refers to a view of a certain set of data (i.e. a list of cars, a list of events, a single car, a single event). A layout is a way that that view is organized.

In our component, we will have a single view called hello, and a single layout (default).

HUBzero Method

Most HUBzero component controllers will differ from Joomla! 1.5 in some important ways. This is, in part, due to legacy issues. Some changes are made to aid in development while others may simply be a difference in philosophy. Note, however, that no differences require hacking or altering Joomla! in any way and HUBzero methodologies will run on any stock Joomla! install.

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

class HelloController extends JObject
{
 private $_name  = NULL;
 private $_data  = array();
 private $_task  = NULL;

 //-----------

 public function __construct( $config=array() )
 {
  $this->_redirect = NULL;
  $this->_message = NULL;
```

```
  $this->_messageType = 'message';

  // Set the controller name
  if (empty( $this->_name )) {
   if (isset($config['name'])) {
    $this->_name = $config['name'];
   } else {
    $r = null;
    if (!preg_match('/(.*)Controller/i', get_class($this), $r)) {
     echo "Controller::__construct() : Can't get or parse class name."
;
    }
    $this->_name = strtolower( $r[1] );
   }
  }

  // Set the component name
  $this->_option = 'com_'.$this->_name;
 }

 //-----------

 public function __set($property, $value)
 {
  $this->_data[$property] = $value;
 }

 //-----------

 public function __get($property)
 {
  if (isset($this->_data[$property])) {
   return $this->_data[$property];
  }
 }

 //-----------

 public function execute()
 {
  $this->_task = JRequest::getString('task', '');

  switch ($this->_task)
  {
   default: $this->hello(); break;
  }
```

```
 }

 //-----------

 public function redirect()
 {
  if ($this->_redirect != NULL) {
   $app =& JFactory::getApplication();
   $app->redirect( $this->_redirect, $this->_message );
  }
 }

 //-----------

 protected function hello()
 {
  // Instantiate a new view
  $view = new JView( array('name'=>'hello') );

  // Pass the view any data it may need
  $view->greeting = 'Hello, World!';

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

  // Output the HTML
  $view->display();
 }
}
```

There appears to be a bit more going on here than in the Joomla! example but both code examples are doing essentially the same thing, as we'll explain.

The first, and most important, difference to note is that we're extending JObject rather than JController. Since we're not employing JController, we need to set up many of our methods manually. Much of it, such as the __constructor and redirect methods are very similar to the Joomla! method--they're simply being established here rather than in JController.

One key difference is how the execute() method is handled. In Joomla! 1.5, any public method is assumed to be an executable task. In the HUBzero method, we're explicitly declaring a list of available tasks and what those tasks execute via the switch statement.

Finally, in our display method, we're instantiating a new view, assigning it some data, and then displaying the output.

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 helper sub-directory of your component directory. Naming convention typically follows a pattern of {ComponentName)Helper(HelperName}. Therefore, our helper class is called HelloHelperOutput.

Here's our com_hello/helpers/output.php helper class:

```
<?php
// No direct access

defined( '_JEXEC' ) or die( 'Restricted access' );

jimport('joomla.application.component.helper');

/**
 * Hello World Component Helper
 *
 * @package    Joomla.Tutorials
 * @subpackage Components
 */
class HelloHelperOutput
{
 /**
  * Method to make all text upper case
  *
   * @access    public
  */
 public 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:

```
class HelloViewHello extends JView
{
 function display($tpl = null)
 {
  include_once(JPATH_COMPONENT.DS.'helpers'.DS.'output.php');

  $greeting = HelloHelperOutput::shout("Hello World");
  $this->assignRef( 'greeting', $greeting );

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

Models

Overview

The concept of model gets its name because this class is intended to represent (or 'model') some entity.

Creating A Model

All Joomla! models extend the JModel class. The naming convention for models in the Joomla! 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 HelloModelHello.

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

jimport( 'joomla.application.component.model' );

/**
 * Hello Model
 */
class HelloModelHello extends JModel
{
    /**
    * Gets the greeting
    * @return string The greeting to be displayed to the user
    */
    function getGreeting()
    {
        return 'Hello, World!';
    }
}
```

You will notice a line that starts with jimport. The jimport function is used to load files from the Joomla! framework that are required for our component. This particular statement will load the file /libraries/joomla/application/component/model.php. The '.'s are used as directory separators and the last part is the name of the file to load. All files are loaded relative to the libraries directory. This particular file contains the class definition for the JModel class, which is necessary because our model extends this class.

Using A Model

The Joomla! framework is setup in such a way that the controller will automatically load the model that has the same name as the view and will push it into the view. We can easily retrieve a reference to our model using the JView::getModel() method. If the model had not followed this convention, we could have passed the model name to JView::getModel().

Here's an example of using a model with our Hello component (com_hello).

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

jimport( 'joomla.application.component.view');

/**
 * HTML View class for the HelloWorld Component
 *
 * @package    HelloWorld
 */

class HelloViewHello extends JView
{
 function display($tpl = null)
 {
  $model = &$this->getModel();
  $greeting = $model->getGreeting();
  $this->assignRef( 'greeting', $greeting );

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

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 JText::_("COM_EXAMPLE_MY_LINE"); ?></p>
```

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

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

See the Languages overview for details.

Layouts

Directory Structures & Files

Views are written in PHP and HTML and have a .php file extension. View scripts are placed in /com_{componentname}/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 default.php.

```
/hubzero
  /components
    /com_{componentname}
      /views
        /{viewname}
          view.html.php
          /tmpl
            default.php
```

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

Creating A View

Joomla! 1.5 Method

The task of the view is very simple: It retrieves the data to be displayed and pushes it into the template. Data is pushed into the template using the JView::assignRef method.

```
<?php

// no direct access
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport( 'joomla.application.component.view');

/**
 * HTML View class for the HelloWorld Component
 *
 * @package    HelloWorld
 */

class HelloViewHello extends JView
```

```
{
    function display($tpl = null)
    {
        $greeting = "Hello World!";
        $this->assignRef( 'greeting', $greeting );

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

HUBzero Method

Not necessary. Data retrieval and template assignment is handled in the controller.

Creating the Template

Joomla! templates/layouts are regular PHP files that are used to layout the data from the view in a particular manner. The variables assigned by the JView::assignRef method can be accessed from the template using $this->{propertyname} (see the template code below for an example).

Our template is very simple: we only want to display the greeting that was passed in from the view - this file is: views/hello/tmpl/default.php:

```
<?php

// No direct access
defined('_JEXEC') or die('Restricted access'); ?>
<h1><?php echo $this->greeting; ?></h1>
```

Routing

Overview

All components in Joomla! 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 "Contacts" component, you could type http://yourhub.org/index.php?option=com_contact.

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/contact, the first portion after the slash translates to the component com_contact. 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.

router.php

Every router.php file has two methods: {ComponentName}BuildRoute() which takes a query string and turns it into a SEF URL and {ComponentName}ParseRoute() which deconstructs a SEF URL back into a query string to be passed to the component.

```
function ExampleBuildRoute(&$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;
}

function ExampleParseRoute($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;
}
```

{ComponentName}BuildRoute()

This method is called when using JRoute::_(). JRoute::_() 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.

```
// $query = 'task=view&id=123&format=rss'
function ExampleBuildRoute(&$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 JRoute::_() which will construct the final SEF URL of example/view/123/rss.

{ComponentName}ParseRoute()

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 {ComponentName}ParseRoute() which then associates each segment with an appropriate variable name based on the segment's position in the array.

```
function ExampleParseRoute($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.

Packaging

Overview

Packaging a component for distribution is relatively easy. 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}
        view.html.php
        /tmpl
          default.php
    /models
      {modelname}.php
    /controllers
      {controllername}.php
  /admin
    {componentname}.php
    controller.php
    /views
      /{viewname}
        view.html.php
        /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 /language/{LanguageName}/{LanguageName}.{ComponentName}.ini and is loaded each time the module is loaded. All of the other files are copied to the /components/{ComponentName} and /administrator/components/{ComponentName}directories of the Joomla! installation.

Modules

Overview

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

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

Directory Structure & Files

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

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

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

/tmpl
: This directory contains template files.
default.php
: This is the module template. This file will take the data collected by mod_{ModuleName}.php and generate the HTML to be displayed on the page.

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

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

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

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

Examples

A simple "Hello, World" module:

Download: Hello World module (.zip)

A module demonstrating database access and language file:

Download: List Names module (.zip)

Installation

Installing

See Installing Extensions for details.

Uninstalling

See Uninstalling Extensions for details.

Manifests

Overview

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

Note: All modules must be prefixed with mod_.

Directory Structure & Files

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

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

Structure

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

A typical module manifest:

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

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

 <!-- Version Date of the Module -->
 <creationDate>2008-06-23</creationDate>
```

```
 <!-- Copyright information -->
 <copyright>All rights reserved by Ambitionality Software LLC 2008.</c
opyright>

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

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

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

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

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

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

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

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

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

```
"" />

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

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

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

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

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

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

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

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

Controllers

Overview

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

Directory Structure & Files

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

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

Implementation

Most modules will perform three tasks in the following order:

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

Here are the contents of mod_listnames.php:

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

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

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

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

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

Helpers

Overview

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

Directory Structure & Files

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

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

Implementation

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

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

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

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

```
}
```

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

Languages

Setup

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

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

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

See the Languages overview for details.

Translating Text

Below is an example of accessing the translate helper:

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

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

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

See the Languages overview for details.

Layouts

Overview

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

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

Directory Structure & Files

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

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

Implementation

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

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

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

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

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

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

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

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

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

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

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

Packaging

Overview

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

Loading

Loading in Templates

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

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

Advanced Template Loading

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Loading in Components

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

XModuleHelper::renderModules($position)

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

XModuleHelper::renderModule($name)

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

output.

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

XModuleHelper::displayModules($position)

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

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

XModuleHelper::renderModule($name)

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

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

Loading in Articles

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

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

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

Plugins

Overview

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

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

See Component Events for a list of Hubzero plugin events.

Core Types

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

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

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

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

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

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

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

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

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

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

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

Directory & File Structure

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

The typical plugin install location and files:

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

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

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

Examples

A plugin demonstrating basic setup:

Download: System Test plugin (.zip)

Installation

Installing

See Installing Extensions for details.

Uninstalling

See Uninstalling Extensions for details.

Manifests

Overview

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

Directory & Files

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

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

Structure

A typical plugin manifest:

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

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

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

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

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

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

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

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

Controllers

Overview

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

Directory & Files

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

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

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

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

Structure

Here we have a typical plugin class:

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

jimport( 'joomla.plugin.plugin' );

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Joomla Events

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

The events triggered in Joomla! are:

Authentication

- onAuthenticate

Content

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

Editors

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

Editors XTD (Extended)

- onDisplay

Search

- onSearch
- onSearchAreas

System

- onAfterInitialise
- onAfterRoute
- onAfterDispatch
- onAfterRender

User

- onLoginUser
- onLoginFailure
- onLogoutUser
- onLogoutFailure
- onBeforeStoreUser

- onAfterStoreUser
- onBeforeDeleteUser
- onAfterDeleteUser

XML-RPC

- onGetWebServices

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

Component Events

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

Groups

- onGroupAreas
- onGroup
- onGroupNew
- onGroupDeleteCount
- onGroupDelete

Members

- onMembersAreas
- onMember

Tools

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

Resources

- onResourcesAreas
- onResources

Support

- onPreTicketSubmission

- onTicketSubmission
- getReportedItem
- deleteReportedItem

Tags

- onTagAreas
- onTagView

Usage

- onUsageAreas
- onUsageDisplay

What's New

- onWhatsnewAreas
- onWhatsnew

XMessage

- onTakeAction
- onSendMessage
- onMessageMethods
- onMessage

XSearch

- onXSearchAreas
- onXSearch

Languages

Overview

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

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

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

Setup

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

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

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

See the Languages overview for details.

Loading

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

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

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

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

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

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

Translating Text

Below is an example of accessing the translate helper:

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

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

See the Languages overview for details.

Layouts

Overview

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

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

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

Directory Structure & Files

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

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

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

Implementation

Loading a plugin view

```
class plgExamplesTest extends JPlugin
{
    ...

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

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

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

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

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

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

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

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

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

The plugin view file

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

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

Packaging

Overview

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

Loading

Triggering Events

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

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

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

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

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

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

Next, we need to trigger our custom event:

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

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

In summary, here's the complete example code:

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

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

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

Templates

Overview

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.

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 PhotoShop design file and finished template that you may use to follow along with the articles or use as a starter for your own HUB template.

Download PhotoShop design file (zip)
Download Basic Template (zip)

Installation

Installing

See Installing Extensions for details.

Uninstalling

See Uninstalling Extensions for details.

Designing

Overview

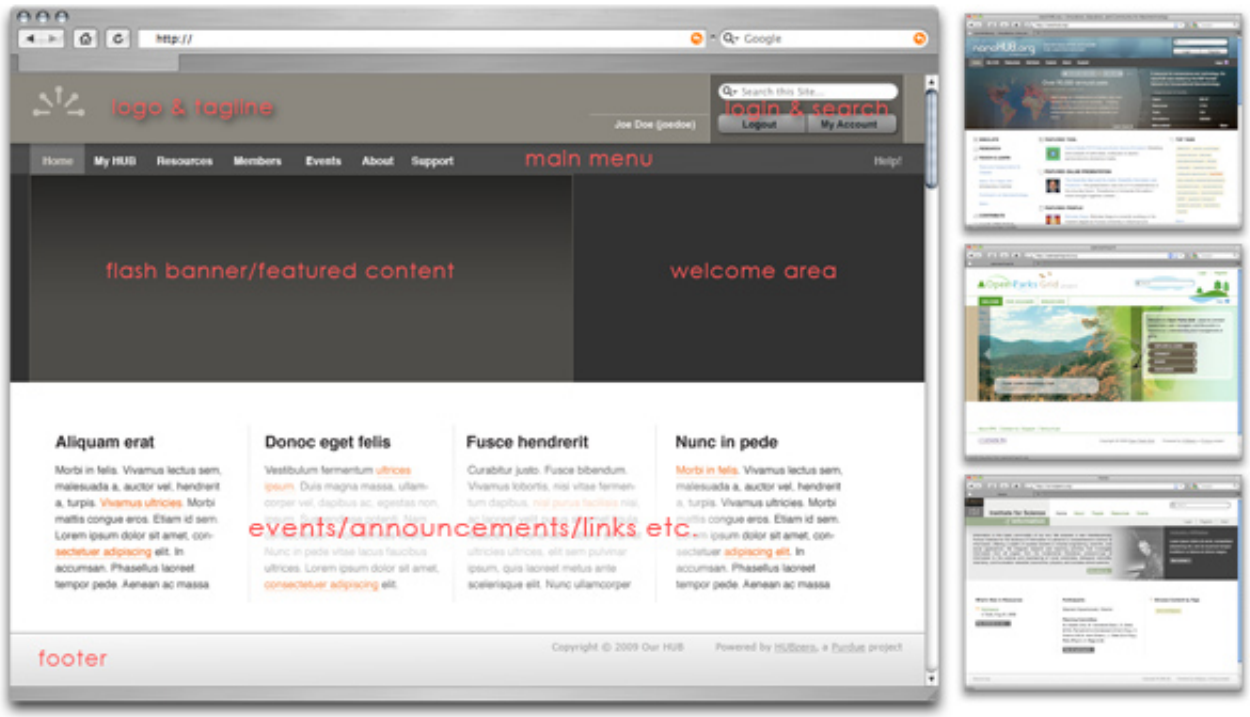
Although many currently available HUBs tend to look somewhat similar, you have the freedom to make your HUB look as unique as you want it to be simply by modifying a few CSS and HTML files within your template folder.

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.

Common elements of a HUB front page & variations



Manifests

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 Joomla!.

Directory & Files

Manifests are stored in the same directory as the template file itself and must be named templateDetails.xml.

```
/hubzero
  /templates
    /{TemplateName}
      /css
      /html
      /images
      /js
      error.php
      index.php
      templateDetails.xml
      template_thumbnail.png
      favicon.ico
```

Structure

This XML file just lines out basic information about the template such as the owner, version, etc. for identification by the Joomla! installer and then provides optional parameters which may be set in the 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 version="1.0" encoding="utf-8"?>
<!DOCTYPE install PUBLIC "-//Joomla! 1.5//DTD template 1.0//EN"
 "http://dev.joomla.org/xml/1.5/template-install.dtd">
<install 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>
</install>
```

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

INSTALL
: 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 params. 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.

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

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
defined( '_JEXEC' ) or die( 'Restricted access' );

// Get the site config
$jconfig =& JFactory::getConfig();
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
 xml:lang="<?php echo $this->language; ?>" lang="<?php echo $this->lan
guage; ?>" >
```

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>
 <!-- This includes metadata tags and the <title> tag -->
 <jdoc:include type="head" />

 <!-- Include the template's main CSS file -->
 <link rel="stylesheet" type="text/css" href="<?php echo $this->baseur
l ?>/templates/<?php echo $this->template; ?>/css/print.css" />
 <!--[if lte IE 7]>
  <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/ie7w
in.css" />
 <![endif]-->
 <!--[if lte IE 6]>
  <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/ie6w
in.css" />
 <![endif]-->
</head>
```

The first 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 print style sheet (if it exist, is named print.css and is located in the

/css folder) and 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 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 . 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 . 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.

Now for the final portion:

```
</html>
<?php
 // Get the page title
 $title = $this->getTitle();
 // Prepend the page title with the site name
 $this->setTitle( $jconfig->getValue('config.sitename').' - '.$title )
;
?>
```

The PHP in the example above is purely optional. What is does is take the current page title and prepends the site's name. Thus, every page results with a title like "myHUB.org - My Page Title".

A Breakdown of error.php

Starting at the top:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

// Get the site config
$jconfig =& JFactory::getConfig();
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
 xml:lang="<?php echo $this->language; ?>" lang="<?php echo $this->lan
guage; ?>" >
```

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.

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
        ie6.css
        ie7.css
        ie8.css
        main.css
        print.css
        reset.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.

ie6.css
: Style fixes for Internet Explorer 6.

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.

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.

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.

```
body,div,dl,dt,dd,ul,ol,li,h1,h2,h3,h4,h5,h6,pre,form,fieldset,input,p
,blockquote,th,td {
 margin:0;
 padding:0;
}
table {
 border-collapse:collapse;
 border-spacing:0;
}
fieldset,img {
 border:0;
}
address,caption,cite,code,dfn,em,strong,th,var {
 font-style:normal;
```

```
 font-weight:normal;
}
ul {
 list-style:none;
}
caption,th {
 text-align:left;
}
h1,h2,h3,h4,h5,h6 {
 font-size:100%;
}
q:before,q:after {
 content:'';
}
```

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.

```
ximport('Hubzero_Document');
Hubzero_Document::addComponentStylesheet('com_example');
```

First, we load the Hubzero_Document 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.

```
ximport('Hubzero_Document');
Hubzero_Document::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.

```
ximport('Hubzero_Document');
Hubzero_Document::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
```

Further Help

Resources for learning and sharpening CSS skills:

- CSS Zen Garden
- CSS From The Ground Up
- Guide to Cascading StyleSheets
- CSS School

Failed to write content for page "JavaScript"

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, "Joomla!" has added 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 "Joomla!" 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.

While Joomla! only allows for overriding views and some HTML, HUBzero has extended this functionality to allow for overriding CSS 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. For example, "rhuk_milkyway" has no component layout overrides at all. When you use this template you are seeing the raw output from all components. When you use the "Beez" template, almost every piece of component output is being controlled by the layout overrides in the template. "corenil" is in between having overrides for some components and only some views of those components.

The layout overrides must be placed in particular way. Using "Beez" as an example you will see the following structure:

```
/templates
  /beez
    /html
      /com_content  (this directory matches the component directory na
me)
```

```
        /articles        (this directory matches the view directory na
me)
          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/rhuk_milkyway/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 na
me)
        com_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:

```
ximport('Hubzero_Document');
Hubzero_Document::addComponentStylesheet('com_example');
```

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 directory
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 directo
ry 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:

```
ximport('Hubzero_Document');
Hubzero_Document::addModuleStylesheet('mod_example');
```

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

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

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 pattern o
f 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
      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 o
f 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:

```
ximport('Hubzero_Document');
Hubzero_Document::addPluginStylesheet('groups', 'forum');
```

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 *
```