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.

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

JavaScript

Overview

Joomla! 1.5 comes with the MooTools 1.11 Javascript Framework. MooTools is not only a visual effects library–it also support Ajax request and JSON notation, table sort, drag & drop operations and much more. All current HUBzero JavaScripts are built on this framework.

Directory & Files

The MooTools framework can be found within the /media/system/js directory. Joomla! includes both a compressed version used for production and an uncompressed version used for debug mode and developer reference.

```
/hubzero
  /media
    /system
      /js
        mootools-uncompressed.js
        mootools.js
```

Most HUBzero templates will include some scripts of their own for basic setup, visual effects, etc. These are generally stored in (but not limited to) a sub-directory, named /js, of the template's main directory.

```
/hubzero
  /templates
    /{TemplateName}
      /js
        globals.js
        hub.js
        modal.js
        tooltips.js
        ...
```

Of the scripts commonly found in a HUBzero template, hub.js and globals.js are perhaps the most important and it is strongly encouraged that developers include these files in their template.

hub.js

If a template includes hub.js, it will be auto-loaded by the system (thus, no reason to specifically declare it in your layout file). When loaded successfully, it will check for the inclusion of the MooTools framework and its version. Should everything pass, the script will then load any other scripts you declare in a comma-separated string.

```
var HUBzero = {
  Version: '1.1',
  require: function(libraryName) {
    // inserting via DOM fails in Safari 2.0, so brute force approach
    document.write('<script type="text/javascript" src="'+libraryName+
'"></script>');
  },
  load: function() {
    if((typeof MooTools=='undefined') ||
      parseFloat(MooTools.version)
      throw("This HUB requires the MooTools JavaScript framework >= 1.
1.0");

    $A(document.getElementsByTagName("script")).each( function(s) {
      if (s.src && s.src.match(/hub.js(?.*)?$/)) {
   var path = s.src.replace(/hub.js(?.*)?$/,'');
      var includes = s.src.match(/?.*load=([a-z,]*)/);
      (includes ? includes[1] : 'globals,tooltips').split(',').each(
       function(include) { HUBzero.require(path+include+'.js') });
   }
    });
  }
}

HUBzero.load();
```

globals.js and the HUB Namespace

Most HUBzero templates will include a global.js file that first establishes a HUB namespace and then proceeds through some basic setup routines. All HUBzero built components, modules, and templates that employ JavaScript place scripts within this HUB namespace. This helps prevent any naming collisions with third-party libraries. While it is recommended that any scripts you may add to your code is also placed within the HUB namespace, it is not required.

Some additional sub-spaces for further organization are available within the HUB namespace.

Separate spaces for Modules, Components, and Plugins are created. Once again, this further helps avoid possible naming/script collisions. Additionally, one more Base space is created for basic setup and utilities that may be used in other scripts.

```
//  Create our namespace
if (!HUB) {
 var HUB = {};

 // Establish a space for setup/init and utilities
 HUB.Base = {};

 // Establish sub-spaces for the various extensions
 HUB.Components = {};
 HUB.Modules = {};
 HUB.Plugins = {};
}
```

To demonstrate adding code to the namespace, below is code from a script in a component named com_example.

```
//  Create our namespace
if (!HUB) {
 var HUB = {};

 // sub-space for components
 HUB.Components = {};
}

// The Example namespace and init method
HUB.Components.Example = {
 init: function() {
  // do something
 }
}

// Initialize the code
window.addEvent('domready', HUB.Components.Example.init);
```

Loading From An Extension

Components

Occasionally a component will have scripts of its own. Pushing JavaScript to the template from a component is quite easy and involves only a few lines of code.

```
// Get the document object
$document =& JFactory::getDocument();
// Check if the file actually exist
if (is_file(JPATH_ROOT.DS.'components'.DS.'com_example'.DS.'example.js
')) {
 // Add the file to the list of scripts to be outputted in the templat
e
 $document->addScript('components'.DS.'com_example'.DS.'example.js');
}
```

First, we load the document object. Next we check for the existence of the JavaScript file we wish to load. If found, we add it to the array of scripts that will be outputted in the <head> of the site template.

Modules

Loading JavaScript from a module is the same as loading from a component save one minor difference: the path to the JavaScript file is obviously different.

```
// Get the document object
$document =& JFactory::getDocument();
// Check if the file actually exist
if (is_file(JPATH_ROOT.DS.'modules'.DS.'mod_example'.DS.'mod_example.j
s')) {
 // Add the file to the list of scripts to be outputted in the templat
e
 $document->addScript('modules'.DS.'mod_example'.DS.'mod_example.js');
}
```

Plugins

Loading JavaScript from a plugin is the same as loading from a component or module save one minor difference: the path to the JavaScript file is obviously different.

```
// Get the document object
$document =& JFactory::getDocument();
// Check if the file actually exist
```

```
if (is_file(JPATH_ROOT.DS.'plugins'.DS.'examples'.DS.'test.js')) {
 // Add the file to the list of scripts to be outputted in the templat
e
 $document->addScript('plugins'.DS.'examples'.DS.'test.js');
}
```

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 *
```