Templates

Overview

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

This article guides you through the process of designing your own template for a HUB. This is intended for web designers/developers with a solid knowledge of CSS and HTML and some basic sense of aesthetics.

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

Note: All the following articles will refer to construction of a front-end template. However, the concepts, techniques, and methods used also apply to the creation of administrative (back-end) templates unless otherwise noted.

Examples

We have provided an example template that you may use to follow along with the articles or use as a starter for your own HUB template.

Download Basic Template (zip)

Structure

Overview

All templates should include a manifest in the form of an XML document named templateDetails.xml. The file holds key "metadata" about the template and is essential. Without it, your template won't be seen by the system.

Directory & Files

Templates are found in the /templates directory of a hub's /app. Specific template files are contained within a directory of the same name as the template. While a template may contain any number of files and sub-directories, it must contain at least two files: the primary layout (index.php) and a XML manifest named templateDetails.xml.

```
/app
.. /templates
.. .. /{TemplateName}
.. .. .. /css
.. .. .. /html
.. .. .. /img
.. .. ..  /js
.. .. .. error.php
.. .. .. component.php
.. .. .. index.php
.. .. .. templateDetails.xml
.. .. .. template_thumbnail.png
.. .. .. favicon.ico
```

Fontcons

Overview

In a single collection, Fontcons is a pictographic language designed for a full array of web-related actions and content. Although originally inspired by Font Awesome, we've heavily modified and added to the available icons; Fontcons brings over 250 icons for use in a package equivalent in file size to just one or two bitmapped icons!

Integration

The open source package contains several bootstrap CSS files for inclusion in your template. These stylesheets can be found in the web root's /media/system/css directory. Here, our attention is on `fontcons.css` which contains the necessary @font-face rules to start using Fontcons.

```
@font-face {
 font-family: 'Fontcons';
 src: url('/media/system/css/fonts/fontcons-webfont.eot');
 src: url('/media/system/css/fonts/fontcons-
webfont.eot?#iefix') format('embedded-opentype'),
   url('/media/system/css/fonts/fontcons-
webfont.woff') format('woff'),
   url('/media/system/css/fonts/fontcons-
webfont.ttf') format('truetype'),
   url('/media/system/css/fonts/fontcons-
webfont.svg#FontconsRegular') format('svg');
 font-weight: normal;
 font-style: normal;
}
```

While you can include Fontcons on a per use basis (e.g., individual components), due to it being relatively light-weight and several Hubzero components making use of it, we recommend including the stylesheet into your site template.

In the <head> of your template's html, reference the location to fontcons.css:

```
<link rel="stylesheet" href="/media/system/css/fontcons.css" />
```

Or import fontcons.css into your site's CSS:

```
/* Note: import rules MUST come first */
@import "/media/system/css/fontcons.css";

/* Other styles here */
```

A word of caution on using @import: Internet Explorer 8 and older will download stylesheets in sequence rather than in parallel. This can have effects on page speed and flashes of un-styled content before the CSS files have finished downloading. See Steve Souder's "donâ€™t use @import" for more details.

Use

There are two primary ways to use the font, both with advantages and disadvantages. The first, is to include the necessary HTML and unicode character directly into your markup.

The HTML:

```
<a href="#"><span class="edit">&#x270E;</span> edit</a>
```

The CSS:

```
.edit {
    font-family: "Fontcons"
}
```

The advantage here is greater browser compatibility. @font-face is supported by even Internet Explorer 6. The disadvantage, however, is that you now have to edit the HTML wherever you wish to insert an icon which could change depending upon the styling and theme of your template. That could quickly become a headache!

The alternative is to use the CSS pseudo-elements :before and :after. This takes a little more setup in your styles but offers greater flexibility and ease of change. Unfortunately, pseudo-elements are **not** supported in Internet Explorer 7 or older. There is, however, a solution which we'll get to in a moment.

The HTML:

```
<a class="edit" href="#">edit</a>
```

The CSS:

```
/* Note the :before pseudo-element */
small.edit,  /* for IE 7, more on that below */
.edit:before {
    font-family: "Fontcons"
    content: "\\270E"; /* unicode characters must start with a backsla
sh */
}
```

What about Internet Explorer 7?

```
.edit {
    *zoom:expression(this.runtimeStyle['zoom']='1', this.innerHTML='<s
mall class="edit">&#x270E;</small>' + this.innerHTML);
}
```

We use <small> in the example above since it's a relatively unused tag and lessens the potential for styling conflicts. It should be noted that over-use of this technique can slow down IE 7 as it has to process and dynamically include content into the page upon render.

Icon List

- \\f000
- \\266B
- \\f002
- \\2709
- \\2665
- \\2605
- \\2606
- \\f007
- \\f008

- \\f009
- \\f00a
- \\f00b
- \\2714
- \\2716
- \\f00e
- \\f010
- \\f011
- \\f012
- \\2699
- \\f014
- \\2302
- \\f016
- \\f017
- \\2641
- \\f01e
- \\f018
- \\f019
- \\f01a
- \\f01b
- \\f01c
- \\f01d
- \\21BB
- \\f083
- \\f092
- \\f085
- \\f08e
- \\f08d
- \\f077
- \\23F0
- \\f071
- \\f081
- \\260E
- \\f056
- \\f067
- \\f062
- \\f044
- \\f061
- \\f069
- \\f07f
- \\f01f
- \\269B
- \\f09c
- \\f095
- \\f0a1
- \\f0a2

- \\f0a3
- \\f0ad
- \\f0ae
- \\f0b0
- \\f0b2
- \\f0e3
- \\f0d0
- \\f0ea

- \\f021
- \\f022
- \\f023
- \\2691
- \\f025
- \\f026
- \\f027
- \\f028
- \\f029
- \\f02a
- \\f02b
- \\f02c
- \\f02d
- \\f02e
- \\2399
- \\f030
- \\f031
- \\f032
- \\f033
- \\f034
- \\f035
- \\f036
- \\f037
- \\f038
- \\f039
- \\f03a
- \\f03b
- \\f03c
- \\f03d
- \\f03e
- \\f082
- \\2692
- \\25F7

- \\f080
- \\f084
- \\26DF
- \\f004
- \\26D3
- \\f00c
- \\237E
- \\f072
- \\231B
- \\f068
- \\f005
- \\f05c
- \\f054
- \\f063
- \\f053
- \\f07d
- \\f07e
- \\f05f
- \\f09a
- \\f08f
- \\f0a4
- \\f0a5
- \\f0a6
- \\f0a7
- \\f0ca
- \\f0cb
- \\f0cc
- \\f0cd
- \\f0ce
- \\f0db

- \\270E
- \\f041
- \\f043
- \\25D1
- \\270D
- \\f045
- \\2611
- \\f047
- \\21E4
- \\f049
- \\219E

- \\25B6
- \\f04c
- \\2588
- \\21A0
- \\21E4
- \\f049
- \\f052
- \\2039
- \\203A
- \\2295
- \\2296
- \\f057
- \\f058
- \\f059
- \\f05a
- \\f05b
- \\2297
- \\f05d
- \\2298
- \\f087
- \\f088
- \\f086
- \\f091
- \\f093
- \\270B
- \\f00d
- \\f08a
- \\f006
- \\f003
- \\f001
- \\f094
- \\f078
- \\f040
- \\f060
- \\f05e
- \\f08c
- \\f079
- \\f097
- \\f098
- \\f03f
- \\f096
- \\f09d
- \\f0a8
- \\f0a9
- \\f0aa
- \\f0ab

- \\f0b1
- \\f0c1
- \\f0c2
- \\f0c3
- \\2622
- \\2746

- \\2190
- \\2192
- \\2191
- \\2193
- \\f064
- \\f065
- \\f066
- \\271A
- \\2010
- \\273D
- \\f06b
- \\f06c
- \\f06d
- \\2601
- \\f046
- \\f06e
- \\f070
- \\26A0
- \\2757
- \\2708
- \\f073
- \\f074
- \\f075
- \\f0e5
- \\f0e6
- \\f02f
- \\2303
- \\2304
- \\267B
- \\f07a
- \\f07b
- \\f07c
- \\2195
- \\2194
- \\f076

TEMPLATES

- \\f090
- \\f08b
- \\f089
- \\2661
- \\26A1
- \\2702
- \\22EF
- \\f055
- \\f042
- \\2693
- \\275D
- \\275E
- \\f04a
- \\f048
- \\f04d
- \\f04e
- \\f06f
- \\f04f
- \\f09b
- \\f0a0
- \\f0d7
- \\f0d8
- \\f0d9
- \\f0da
- \\f0d6
- \\f0ea
- \\f0c5

Socicons

Overview

In a single collection, Socicons is a pictographic language containing icons for some of the most popular social and web services such as Twitter, Facebook, and Google.

Integration

The open source package contains several bootstrap CSS files and fonts for inclusion in your template. Below is the necessary @font-face rules to start using Socicons.

```
@font-face {
 font-family: 'Socicons';
 src: url('/media/system/css/fonts/socicons-webfont.eot');
 src: url('/media/system/css/fonts/socicons-
webfont.eot?#iefix') format('embedded-opentype'),
   url('/media/system/css/fonts/socicons-
webfont.woff') format('woff'),
   url('/media/system/css/fonts/socicons-
webfont.ttf') format('truetype'),
   url('/media/system/css/fonts/socicons-
webfont.svg#SociconsRegular') format('svg');
 font-weight: normal;
 font-style: normal;
}
```

Socicons is relatively lightweight due to the limited number of icons available and can be either included in the stylesheet into your site template or on a per use basis (e.g., individual components).

Use

There are two primary ways to use the font, both with advantages and disadvantages. The first, is to include the necessary HTML and unicode character directly into your markup.

The HTML:

```
<a href="#"><span class="facebook">&#xf013;</span> facebook</a>
```

The CSS:

```
.facebook {
    font-family: "Socicons"
}
```

The advantage here is greater browser compatibility. @font-face is supported by even Internet Explorer 6. The disadvantage, however, is that you now have to edit the HTML wherever you wish to insert an icon which could change depending upon the styling and theme of your template. That could quickly become a headache!

The alternative is to use the CSS pseudo-elements :before and :after. This takes a little more setup in your styles but offers greater flexibility and ease of change. Unfortunately, pseudo-elements are **not** supported in Internet Explorer 7 or older. There is, however, a solution which we'll get to in a moment.

The HTML:

```
<a class="facebook" href="#">facebook</a>
```

The CSS:

```
/* Note the :before pseudo-element */
small.facebook,  /* for IE 7, more on that below */
.facebook:before {
    font-family: "Socicons"
    content: "\\f013"; /* unicode characters must start with a backsla
sh */
}
```

What about Internet Explorer 7?

```
.facebook {
    *zoom:expression(this.runtimeStyle['zoom']='1', this.innerHTML='<s
mall class="facebook">&#xf013;</small>' + this.innerHTML);
}
```

We use <small> in the example above since it's a relatively unused tag and lessens the potential for styling conflicts. It should be noted that over-use of this technique can slow down IE 7 as it has to process and dynamically include content into the page upon render.

Icon List

- \\f002 Hub
- \\f001 Hub alt
- \\f006 Purdue
- \\f005 Purdue alt
- \\f013 Facebook
- \\f012 Facebook alt
- \\f026 Dropbox
- \\f025 Dropbox alt

- \\f011 Twitter
- \\f010 Twitter alt
- \\f019 Github
- \\f018 Github alt
- \\f024 PayPal
- \\f023 PayPal alt
- \\f02a eBay
- \\f029 eBay alt

- \\f017 LinkedIn
- \\f016 LinkedIn alt
- \\f01b Pinterest
- \\f01a Pinterest alt
- \\f022 Skype
- \\f021 Skype alt
- \\f028 Dribbble
- \\f027 Dribbble alt

- \\f02c Google
- \\f02b Google alt
- \\f015 Google+
- \\f014 Google+ alt
- \\f01d Vimeo
- \\f01e Vimeo alt
- \\f01f YouTube
- \\f01e YouTube alt

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. Override files must be placed in folders in the folder "html".

```
/{TemplateName}
  /css
    ... CSS files ...
  /html
    ... Overrides ...
  /images
    ... Image files ...
  /js
    ... JavaScript files ...
  composer.json
  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

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

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

- optionally, a destined install directory

Composer Manifest

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

A typical component manifest:

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

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

XML Manifest (deprecated)

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

A typical template manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<extension version="1.5" type="template">
 <name>mynewtemplate</name>
 <creationDate>2008-05-01</creationDate>
 <author>John Doe</author>
 <authorEmail>john@example.com</authorEmail>
 <authorUrl>http://www.example.com</authorUrl>
 <copyright>John Doe 2008</copyright>
 <license>GNU/GPL</license>
 <version>1.0.2</version>
 <description>My New Template</description>
 <files>
  <filename>index.php</filename>
  <filename>component.php</filename>
  <filename>templateDetails.xml</filename>
  <filename>template_thumbnail.png</filename>
  <filename>images/background.png</filename>
  <filename>css/style.css</filename>
 </files>
 <positions>
  <position>breadcrumb</position>
  <position>left</position>
  <position>right</position>
  <position>top</position>
  <position>user1</position>
  <position>user2</position>
  <position>user3</position>
  <position>user4</position>
  <position>footer</position>
 </positions>
</extension>
```

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

EXTENSION
    The install tag has several key attributes. The type must be "template".
NAME
    You can name the templates in any way you wish.
FILES
    The files tag includes all of the files that will will be installed with the template.
POSITIONS
    The module positions used in the template.

The one noticeable difference between this template manifest and the typical manifest of a module or component is the lack of config. While templates may have their own params for further configuration via the administrative back-end, they aren't as commonly found as in other extension manifests. Most HUBzero templates do not include them.

Output Overrides

Overview

There are many competing requirements for web designers ranging from accessibility to legislative to personal preferences. Rather than trying to over-parameterise views, or trying to aim for some sort of line of best fit, or worse, sticking its head in the sand, the CMS gives the potential for the designer to take over control of virtually all of the output that is generated.

Except for files that are provided in the distribution itself, these methods for customization eliminate the need for designers and developers to "hack" core files that could change when the site is updated to a new version. Because they are contained within the template, they can be deployed to the Web site without having to worry about changes being accidentally overwritten when your System Administrator upgrades the site.

HUBzero allows for overriding not only views but CSS and Javascript as well. This allows for even more individualistic styling of components and modules on HUBs.

Component Overrides

Note: Not all HUBzero modules will have layouts or CSS that can be overridden.

Layouts

Layout overrides only work within the active template and are located under the /html/ directory in the template. For example, the overrides for "corenil" are located under /app/templates/corenil/html/.

It is important to understand that if you create overrides in one template, they will not be available in other templates.

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

```
/templates
.. /kimera
.. .. /html
.. .. .. /com_content  (this directory matches the component directory
 name)
.. .. .. .. /articles        (this directory matches the view director
y name)
.. .. .. .. .. default.php (this file matches the layout file name)
.. .. .. .. .. form.php
```

The structure for component overrides is quite simple:
/html/com_{ComponentName}/{ViewName}/{LayoutName}.php.

Sub-Layouts

In some views you will see that some of the layouts have a group of files that start with the same name. The category view has an example of this. The blog layout actually has three parts: the main layout file blog.php and two sub-layout files, blog_item.php and blog_links.php. You can see where these sub-layouts are loaded in the blog.php file using the loadTemplate method, for example:

```
echo $this->loadTemplate('item');
// or
echo $this->loadTemplate('links');
```

When loading sub-layouts, the view already knows what layout you are in, so you don't have to provide the prefix (that is, you load just 'item', not 'blog_item').

What is important to note here is that it is possible to override just a sub-layout without copying the whole set of files. For example, if you were happy with the default output for the blog layout, but just wanted to customize the item sub-layout, you could just copy:

```
/components/com_content/views/category/tmpl/blog_item.php
```

to:

```
/templates/kimera/html/com_content/category/blog_item.php
```

When the CMS is parsing the view, it will automatically know to load blog.php from com_content natively and blog_item.php from your template overrides.

Cascading Style Sheets

Over-ridding CSS is a little more straight-forward over-ridding layouts. Take the com_groups component for example:

```
/components
  /com_groups
    ...
    com_groups.css   (the component CSS file)
```

To override the CSS, we simply copy or create a new CSS file named the same and place it in the template's overrides:

```
/templates
.. /corenil
.. .. /html
.. .. .. /com_groups   (this directory matches the component directory
 name)
.. .. .. .. groups.css   (this file matches the CSS file name)
```

To push CSS from a component to the template, add the following somewhere in the component:

```
$this->css('example.css');
```

Module Overrides

Note: Not all HUBzero modules will have layouts or CSS that can be overridden.

Layouts

Modules, like components, are set up in a particular directory structure.

```
/modules
.. /mod_latest_news
.. .. /tmpl
.. .. .. default.php   (the layout)
.. .. .. helper.php   (a helper file containing data logic)
.. .. mod_latest_news.php   (the main module file)
.. .. mod_latest_news.xml   (the installation XML file)
```

Similar to components, under the main module directory (in the example, mod_latest_news) there is a /tmpl/ directory. There is usually only one layout file but depending on who wrote the module, and how it is written, there could be more.

As for components, the layout override for a module must be placed in particular way. Using "corenil" as an example again, you will see the following structure:

```
/templates
.. /corenil
.. .. /html
.. .. .. /mod_latest_news   (this directory matches the module directo
ry name)
.. .. .. .. default.php   (this file matches the layout file name)
```

Take care with overriding module layout because there are a number of different ways that modules can or have been designed so you need to treat each one individually.

Cascading Style Sheets

Over-ridding CSS files works in precisely the same way as over-ridding layouts. Take the mod_reportproblems module for example:

```
/modules
  /mod_reportproblems
    ...
    mod_reportproblems.css   (the module CSS file)
```

To override the CSS, we simply copy or create a new CSS file named the same and place it in the template's overrides:

```
/templates
.. /corenil
.. .. /html
.. .. .. /mod_reportproblems   (this directory matches the module dire
ctory name)
.. .. .. .. mod_reportproblems.css
   (this file matches the CSS file name)
```

To push CSS from a module to the template, add the following somewhere in the module:

```
$this->css('mod_example.css');
```

Plugin Overrides

Note: Not all HUBzero plugins will have layouts or CSS that can be overridden.

Layouts

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

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

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

As with components and modules, the layout override for a plugin must be placed in a particular way. Using "corenil" as an example again, you will see the following structure:

```
/templates
.. /corenil
.. .. /html
.. .. .. /plg_groups_forum   (this directory follows the naming patter
n of plg_{group}_{plugin})
.. .. .. .. /browse   (this file matches the layout directory name)
.. .. .. .. .. default.php   (this file matches the layout file name)
```

Take care with overriding plugin layout because there are a number of different ways that plugins can or have been designed so you need to treat each one individually.

Cascading Style Sheets

Over-ridding CSS files works in precisely the same way as over-ridding layouts. Take the forum plugin for groups for example:

```
/plugins
.. /groups
.. .. /forum
.. .. .. /assets
.. .. .. .. /css
.. .. .. .. .. forum.css   (the plugin CSS file)
```

To override the CSS, we simply copy or create a new CSS file named the same and place it in the template's overrides:

```
/templates
.. /corenil
.. .. /html
.. .. .. /plg_groups_forum   (this directory follows the naming patter
n of plg_{group}_{plugin})
.. .. .. .. forum.css   (this file matches the CSS file name)
```

To push CSS from a module to the template, add the following somewhere in the module:

```
$this->css('forum.css');
```

Pagination Links Overrides

This override can control the display of items-per-page and the pagination links that are used with lists of information. Most HUBzero templates will come with a pagination override that outputs what we feel is a good standard for displaying pagination links and controls. However, feel free to alter this as you see fit. The override can be found here:

```
/templates/{TemplateName}/html/pagination.php
```

When the pagination list is required, Joomla! will look for this file in the default templates. If it is found it will be loaded and the display functions it contains will be used. There are four functions that can be used:

pagination_list_footer
: This function is responsible for showing the select list for the number of items to display per page.

pagination_list_render
: This function is responsible for showing the list of page number links as well at the Start, End, Previous and Next links.

pagination_item_active
: This function displays the links to other page numbers other than the "current" page.

pagination_item_inactive
: This function displays the current page number, usually not hyperlinked.

Quick Reference

Using the corenil template as an example, here is a brief summary of the principles that have been discussed.

Note: Not all HUBzero components, plugins, and modules will have layouts that can be overridden.

Component Output

To override a component layout (for example the default layout in the article view), copy:

```
/components/com_content/views/article/tmpl/default.php
```

to:

```
/templates/corenil/html/com_content/article/default.php
```

To override a component CSS (for example the stylesheet in the com_groups), copy:

```
/components/com_groups/site/assets/css/com_groups.css
```

to:

```
/templates/corenil/html/com_groups/groups.css
```

To push CSS from a component to the template, add the following somewhere in the component:

```
HubzeroDocumentAssets::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/assets/css/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:

```
HubzeroDocumentAssets::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/assets/css/forum.css
```

To push CSS from a plugin to the template, add the following somewhere in the plugin:

```
HubzeroDocumentAssets::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
```

JavaScript

Overview

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

Directory & Files

The jQuery framework can be found within the /core/assets/js directory. It is a compressed version used for production. An uncompressed version may be found at jquery.com.

```
/hubzero
  /media
    /system
      /js
        jquery.js
```

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

```
/hubzero
  /media
    /system
      /js
        jquery.fancybox.js
        jquery.fileuploader.js
        jquery.ui.js
```

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

hub.js

```
//-------------------------------------------------------------
//  Create our namespace
//-------------------------------------------------------------
var HUB = HUB || {};
HUB.Base = {};

var alertFallback = true;
if (typeof console === "undefined" || typeof console.log === "undefine
d") {
 console = {};
 console.log = function() {};
}

//-------------------------------------------------------------
//  Various functions - encapsulated in HUB namespace
//-------------------------------------------------------------
if (!jq) {
 var jq = $;

 $.getDocHeight = function(){
  var D = document;
  return Math.max(Math.max(D.body.scrollHeight, D.documentElement.scro
llHeight), Math.max(D.body.offsetHeight, D.documentElement.offsetHeigh
t), Math.max(D.body.clientHeight, D.documentElement.clientHeight));
 };
} else {
 jq.getDocHeight = function(){
  var D = document;
  return Math.max(Math.max(D.body.scrollHeight, D.documentElement.scro
llHeight), Math.max(D.body.offsetHeight, D.documentElement.offsetHeigh
t), Math.max(D.body.clientHeight, D.documentElement.clientHeight));
 };
}

var template = {};

jQuery(document).ready(function(jq){
 var $ = jq,
  w = 760,
  h = 520,
  templatepath = '/templates/template/';

 // Set focus on username field for login form
 if ($('#username').length > 0) {
  $('#username').focus();
 }

 // Turn links with specific classes into popups
 $('a').each(function(i, trigger) {
  if ($(trigger).is('.demo, .popinfo, .popup, .breeze')) {
   $(trigger).on('click', function (e) {
    e.preventDefault();

    if ($(this).attr('class')) {
     var sizeString = $(this).attr('class').split(' ').pop();
     if (sizeString && sizeString.match(/d+xd+/)) {
      var sizeTokens = sizeString.split('x');
      w = parseInt(sizeTokens[0]);
      h = parseInt(sizeTokens[1]);
     }
     else if(sizeString && sizeString == 'fullxfull')
     {
      w = screen.width;
      h = screen.height;
     }
    }

    window.open($(this).attr('href'), 'popup', 'resizable=1,scrollbars
=1,height='+ h + ',width=' + w);
   });
  }
  if ($(trigger).attr('rel') && $(trigger).attr('rel').indexOf('extern
al') !=- 1) {
   $(trigger).attr('target', '_blank');
  }
 });

 if (jQuery.fancybox) {
  // Set the overlay trigger for launch tool links
  $('.launchtool').on('click', function(e) {
   $.fancybox({
    closeBtn: false,
    href: templatepath + 'images/anim/circling-ball-loading.gif'
   });
  });

  // Set overlays for lightboxed elements
  $('a[rel=lightbox]').fancybox();
 }
 // Init tooltips
 if (jQuery.ui && jQuery.ui.tooltip) {
  $(document).tooltip({
   items: '.hasTip, .tooltips',
   position: {
    my: 'center bottom',
    at: 'center top'
   },
   // When moving between hovering over many elements quickly, the too
ltip will jump around
   // because it can't start animating the fade in of the new tip unti
l the old tip is
   // done. Solution is to disable one of the animations.
   hide: false,
   content: function () {
    var tip = $(this),
     tipText = tip.attr('title');

    if (tipText.indexOf('::') != -1) {
     var parts = tipText.split('::');
     tip.attr('title', parts[1]);
    }
    return $(this).attr('title');
   },
   tooltipClass: 'tooltip'
  });

  // Init fixed position DOM: tooltips
  $('.fixedToolTip').tooltip({
   relative: true
  });
 }

 //test for placeholder support
 var test = document.createElement('input'),
  placeholder_supported = ('placeholder' in test);

 //if we dont have placeholder support mimic it with focus and blur ev
ents
 if (!placeholder_supported) {
  $('input[type=text]:not(.no-legacy-placeholder-
support)').each(function(i, el) {
   var placeholderText = $(el).attr('placeholder');

   //make sure we have placeholder text
   if (placeholderText != '' && placeholderText != null) {
    //add plceholder text and class
    if ($(el).val() == '') {
     $(el).addClass('placeholder-support').val(placeholderText);
    }

    //attach event listeners to input
    $(el)
     .on('focus', function() {
      if ($(el).val() == placeholderText) {
       $(el).removeClass('placeholder-support').val('');
      }
     })
     .on('blur', function(){
      if ($(el).val() == '') {
       $(el).addClass('placeholder-support').val(placeholderText);
      }
     });
   }
  });

  $('form').on('submit', function(event){
   $('.placeholder-support').each(function (i, el) {
    $(this).val('');
   });
  });
 }
};
```

HUB Namespace

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

Note: When not using jQuery, the template will include a global.js file that establishes the HUB namespace.

Some additional sub-spaces for further organization are available within the HUB namespace. Separate spaces for Modules, Components, and Plugins are created. Once again, this further helps avoid possible naming/script collisions. Additionally, one more Base space is created for basic setup and utilities that may be used in other scripts.

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

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

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

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

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

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

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

// Initialize the code (jQuery)
jQuery(document).ready(function($){
 Components.Example.init();
});
```

Loading From An Extension

Components

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

```
HubzeroDocumentAssets::addComponentScript('com_example');
```

First, we load the HubzeroDocumentAssets class. Next we call the static method addComponentScript, passing it the name of the component as the first (and only) argument. This will first check for the presence of the style sheet in the active template's overrides. If found, the path to the overridden script will be added to the array of scripts the template needs to include in the <head>. If no override is found, the code then checks for the existence of the script in the component's directory. Once again, if found, it gets pushed to the template.

Modules

Loading Javascript from a module works virtually the same as loading from a component save one minor difference in code. Instead of calling the addComponentScript method, we call the addModuleScript method and pass it the name of the module.

```
HubzeroDocumentAssets::addModuleScript('mod_example');
```

Plugins

Loading Javascript from a plugin works similarly to loading from a component or module but instead we call the addPluginScript method and pass it the name of the plugin group **and** the name of the plugin.

```
HubzeroDocumentAssets::addPluginScript('examples', 'test');
```

Plugin Javascript must be named the same as the plugin and located within a directory of the same name as the plugin inside the plugin group directory.

```
/plugins
  /examples
    /test
      test.css
    test.php
    test.xml
```

View Helpers (all extensions)

Modules, Component, and plugin views now have helpers for pushing Cascading StyleSheets and JavaScript assets to the document. Each method automatically looks for overrides within the current, active template, taking out the busy work of checking yourself each time assets are added. The method names are short, accept a range of options, and allow for method chaining, all tailored for brevity and ease of use.

The css() method provides a quick and convenient way to attach stylesheets. For components, it accepts two arguments:

1. The name of the stylesheet to be pushed to the document (file extension is optional). If no name is provided, the name of the component or plugin will be used. For instance, if called within a view of the component "com_tags", the system will look for a stylesheet named "tags.css".
2. The name of the extension to look for the stylesheet. For components, this will be the component name (e.g., com_tags). For plugins, this is the name of the plugin folder and requires the third argument of plugin group (type) be passed to the method.
3. *Plugin views only.* The name of the plugin.

Example:

```
<?php
// Push a stylesheet to the document
$this->css()
      ->css('another')  // Extension (.css) is optional
      ->css('tags.css', 'com_tags');  // Load CSS from another compone
nt
?>
... view HTML ...
```

Along with file names, the method also accepts style declarations:

```
<?php
// Push a stylesheet to the document
$this->css('.foo {
 color: #000;
}');
?>
... view HTML ...
```

Similarly, a js() method is available for pushing javascript assets to the document. The arguments accepted are the same as the css() method described above.

```
<?php
// Push some javascript to the document
$this->js()
      ->js('another');
?>
... view HTML ...
```

And, just as the css() method accepts style declarations, the js() method accepts script declarations:

```
<?php
// Push some javascript to the document
$this->js('
 jQuery(document).ready(function($){
  $("a").on("click", function(e){
   console.log($(this).attr("href"));
  });
 });
');
?>
... view HTML ...
```

Cascading Style Sheets

Overview

CSS stands for Cascading Style Sheet. HTML tags specify the graphical flow of the elements, be it text, images or flash animations, on a webpage. CSS allows us to define the appearances of those HTML tags with their content, somewhere, so that other pages, if want be, may adhere to. This brings along consistency throughout a website. The cascading effect stipulates that the style of a tag (parent) may be inherited by other tags (children) inside it.

Professional websites separate styling from content. There are many reasons for this, the most obvious (to a developer) being the ability to control the appearance of many pages by changing one file. Styling information includes: fonts, backgrounds, images (that recur on every page), position and dimensions of elements on the page. Your HTML file will now be left with: header information; a series of elements; the text of your website. Because you are creating a Joomla! template, you will actually have: some header information, PHP code to request the rest of the header information, a series of elements, PHP code to request each module position, and PHP code to request the main content.

Style information is coded in CSS and usually stored in files with the suffix .css. A webpage contains a link to the associated .css file so a browser can find the appropriate style information to apply to the page. CSS can also be placed inside a HTML file between <style type="text/css"></style> tags. This is, however, discouraged as it is mixing style and content elements which can make future changes more difficult.

Implementation

Definitions for this section:

External CSS files
- using <link> in the <head>
- Document head CSS
  - using <style> in the <head>
- Inline CSS
  - using the style attribute on a tag, i.e. <div style="color:red;">

Guidelines

1. External CSS files should be used in preference to document head CSS and document head CSS should be used in preference to inline CSS.
2. CSS files MUST have the file extension .css and should be stored in the relevant includes directory in the site structure, usually /style/.
3. The file size of CSS files should be kept as low as possible, especially on high demand pages.
4. External CSS must be linked to using the <link> element which must be placed

in the head section of the document. This is the preferred method of using CSS. It offers the best experience for the user as it helps prevent FOUC (flash of unstyled content), promotes code reuse across a site and is cacheable.

5. External style sheets should not be imported (i.e. using @import) as it impairs caching. In IE @import behaves the same as using <link> at the bottom of the page (preventing progressive rendering), so it's best not to use it. Mixing <link> and @import has a negative effect on browsers' ability to asynchronously download the files.
6. Document head CSS may be used where a style rule is only required for a specific page.
7. Inline styles should not be used.
8. Query string data (e.g. "style.css?v=0.1") should not be used on an external CSS file. Use of query strings on CSS files prevents them from caching in some browsers. Whilst this may be desirable for testing, and of course may be used for that, it is very undesirable for production sites.

Directory & Files

Convention places CSS files within a directory named css inside the template directory. While developers are not restricted to this convention, we do recommend it as it helps keep the layout and structure of HUBzero templates consistent. A developer from one project will instantly know where to find certain files and be familiar with the directory structure when working on a project originally developed by someone else.

There are a handful of common CSS files found among most HUBzero. While none of these are required, it is encouraged to follow the convention of including them as it promotes consistency among HUBzero templates and comes with the advantage that certain files, such as main.css are auto-loaded, thus reducing some work on the developer's part.

Here's the standard directory and files for CSS found in a HUBzero template:

```
/hubzero
  /templates
    /{TemplateName}
      /css
        error.css
        browser/ie7.css
        browser/ie8.css
        browser/ie9.css
        main.css
        print.css
        component.css
```

File details:

error.css	This is the primary stylesheet loaded by error.php.
ie8.css	Style fixes for Internet Explorer 8.
ie7.css	Style fixes for Internet Explorer 7.
ie9.css	Style fixes for Internet Explorer 9.
main.css	This is the primary stylesheet loaded by index.php. The majority of your styles will be in here.
print.css	Styles used when printing a page.
component.css	This file is meant to be included **before** any other CSS file. Its purpose is to reduce browser inconsistencies in things like default line heights, margins and font sizes of headings, and so on.

Bootstrap

Several bootstrap styles are available in the core, broken into individual stylesheets to make it easier for you to decide what styles you do and do not want to incorporate into your template.

The bootstrap stylesheets can be found in the /core/assets/css directory and can be linked to or imported like any other stylesheet. However, for sake of site performance, we recommend using the HubzeroDocumentAssets::getSystemStylesheet() method. This method accepts wither a comma-separated string or array of core stylesheets to include and then compiles them into a single file with comments and white-space stripped out. The resulting file is saved in the cache with a timestamp. Should any of the core files change, the resulting compiled stylesheet will automatically be updated. This has two immediate advantages of 1) fewer http requests (improves page load time) and 2) ensures browsers re-cache the CSS whenever it has changed.

Example usage:

```
<link rel="stylesheet" type="text/css" media="screen" href="<?php
 echo HubzeroDocumentAssets::getSystemStylesheet(array(
 'reset',
 'fontcons',
 'columns',
 'notifications',
 'pagination',
 'tabs',
 'tags',
 'comments',
 'voting',
 'layout'
)); ?>" />
```

reset.css

This file is meant to be included **before** any other CSS file. Its purpose is to reduce browser inconsistencies in things like default line heights, margins and font sizes of headings, and so on.

The reset styles given here are intentionally very generic. There isn't any default color or background set for the <body> element, for example. Colors and any other styling should be addressed in the template's primary stylesheet after loading reset.css.

fontcons.css

This is a custom created icon (dingbat) font used for many of the icons found throughout a hub.

columns.css

This sets up basic structure for generating layouts that use columns. It supports up to twelve columns and any combination there in. See usage.

notifications.css

Default styles for warning, error, help, and info messages.

pagination.css

Basic styling for pagination.

tabs.css

Default styles for a menu (list) displayed as tabs.

tags.css

Tag styles. Tags are used frequently throughout a hub and this stylesheet helps ensure the look consistent.

comments.css

Comments appear on many items such as KB articles, Questions and Answers, Support tickets, Forums, Blog posts, and more. This is a stylesheet for handling basic layout and styles of a list of (nested) comments and the form for submitting comments.

voting.css

Basic styles for thumbs-up and thumbs-down voting buttons.

layout.css

Default styles for containers, result lists, and other basic structural items used frequently in a hub.

Typical main.css Structure

main.css controls base styling for your HUB, which is usually further extended by individual component CSS.

We took every effort to organize the main.css in a manner allowing you to easily find a section and a class name to modify. E.g. if you want to change the way headers are displayed, look for "headers" section as indicated by CSS comments. Although you can modify all existing classes, depending on your objectives, it is recommended to avoid modifications to certain sections, as indicated below. While you can add new classes as needed, we caution strongly about removing or renaming any of the existing IDs and classes. Many HUBzero components take advantage of these code styles and any alterations made risk breaking the template display.

Some sections that you are likely to modify:

```
Body - may want to change site background or font family.
Links - pick colors for hyperlinks
Headers - pick colors and font size of headings
```

```
Lists - may want to change general list style
Header - you will definitely want to change this
Toolbar - display of username, login/logout links etc.
Navigation - display of main menu
Breadcrumbs - navigation under menu on secondary pages
Extra nav - links that appear on the right-
hand side in multiple components
Footer
```

Sections where you would want to avoid serious modifications:

```
Core classes
Site notices, warnings, errors
Primary Content Columns
Flexible Content Columns
Sub menu - display of tabs in multiple components
```

print.css

This is a style sheet that is used only for printing. It removes unnecessary elements such as menus and search boxes, adjusts any background and font colors as needed to improve readability, and can expose link URLs through generated content (advanced browsers only, e.g. Safari, Firefox).

error.css

This is a style sheet that is used only by the error.php layout. It allows for a more custom styling to error pages such as "404 - Page Not Found".

Internet Explorer

We strongly encourage developers to test their templates in as many browsers and on as many operating systems as possible. Most modern browsers will have little differences in rendering, however, Internet Explorer deserves special mention here.

The most widely used browser, Internet Explorer, is also one of the most lacking in terms of CSS support. Internet Explorer has also, traditionally, handled rendering of block elements, element positioning, and other common tasks a bit differently than many other browsers. As can be expected, this has led to much controversy and discussion on how best to handle such differences. We strongly recommend designing for and testing your templates in alternate browsers such as Safari, Firefox, Chrome, or Opera and then applying fixes to Internet Explorer afterwards. We recommend the use of conditional comments to apply special Internet Explorer only stylesheets.

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 l
ower than 6
<![endif]-->
<!--[if lte IE 5.5]>
 According to the conditional comment this is Internet Explorer l
ower or equal to 5.5
<![endif]-->
<!--[if gt IE 6]>
 According to the conditional comment this is Internet Explorer g
reater 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=
"{TemplatePath}/{TemplateName}/css/ie7.css" />
    <![endif]-->
    <!--[if lte IE 6]>
      <link rel="stylesheet" type="text/css" media="screen" href=
"{TemplatePath}/{TemplateName}/css/ie6.css" />
    <![endif]-->
  </head>
  ...
</html>
```

Note: Conditional comments used CSS for should be placed inside the <head> tag of a template *after* all other CSS have been linked for their affects to properly take place.

Loading From An Extension

Components

Often a component will have a style sheet of its own. Pushing CSS to the template from a component is quite easy and involves only two lines of code.

```
HubzeroDocumentAssets::addComponentStylesheet('com_example');
```

First, we load the HubzeroDocumentAssets class. Next we call the static method addComponentStylesheet, passing it the name of the component as the first (and only) argument. This will first check for the presence of the style sheet in the active template's overrides. If found, the path to the overridden style sheet will be added to the array of style sheets the template needs to include in the <head>. If no override is found, the code then checks for the existence of the CSS in the component's directory. Once again, if found, it gets pushed to the template.

Modules

Loading CSS from a module works virtually the same as loading from a component save one minor difference in code. Instead of calling the addComponentStylesheet method, we call the addModuleStylesheet method and pass it the name of the module.

```
HubzeroDocumentAssets::addModuleStylesheet('mod_example');
```

Plugins

Loading CSS from a plugin works similarly to loading from a component or module but instead we call the addPluginStylesheet method and pass it the name of the plugin group **and** the name of the plugin.

```
HubzeroDocumentAssets::addPluginStylesheet('examples', 'test');
```

Plugin CSS must be named the same as the plugin and located within a directory of the same name as the plugin inside the plugin group directory.

```
/plugins
  /examples
    /test
      test.css
    test.php
    test.xml
```

View Helpers (all extensions)

Modules, Component, and plugin views now have helpers for pushing Cascading StyleSheets and JavaScript assets to the document. Each method automatically looks for overrides within the current, active template, taking out the busy work of checking yourself each time assets are added. The method names are short, accept a range of options, and allow for method chaining, all tailored for brevity and ease of use.

The css() method provides a quick and convenient way to attach stylesheets. For components, it accepts two arguments:

1. The name of the stylesheet to be pushed to the document (file extension is optional). If no name is provided, the name of the component or plugin will be used. For instance, if called within a view of the component "com_tags", the system will look for a stylesheet named "tags.css".
2. The name of the extension to look for the stylesheet. For components, this will be the component name (e.g., com_tags). For plugins, this is the name of the plugin folder and requires the third argument of plugin group (type) be passed to the method.

3. *Plugin views only.* The name of the plugin.

Example:

```
<?php
// Push a stylesheet to the document
$this->css()
      ->css('another')  // Extension (.css) is optional
      ->css('tags.css', 'com_tags');  // Load CSS from another co
mponent
?>
... view HTML ...
```

Along with file names, the method also accepts style declarations:

```
<?php
// Push a stylesheet to the document
$this->css('.foo {
 color: #000;
}');
?>
... view HTML ...
```

Similarly, a js() method is available for pushing javascript assets to the document. The arguments accepted are the same as the css() method described above.

```
<?php
// Push some javascript to the document
$this->js()
      ->js('another');
?>
... view HTML ...
```

And, just as the css() method accepts style declarations, the js() method accepts script declarations:

```
<?php
```

```
// Push some javascript to the document
$this->js('
 jQuery(document).ready(function($){
  $("a").on("click", function(e){
   console.log($(this).attr("href"));
  });
 });
');
?>
... view HTML ...
```

Further Help

Resources for learning and sharpening CSS skills:

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

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).

```
/app
  /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( '_HZEXEC_' ) or die( 'Restricted access' );

$this->addScript($this->baseurl . '/templates/' . $this->template . '/
js/hub.js');

// Get the user's browser and browser version
// We add this to the document root as classes for better targeting wi
```

```
th CSS
$browser = new HubzeroBrowserDetector();
$b = $browser->name();
$v = $browser->major();

// Set the page title
$this->setTitle(Config::get('sitename') . ' - ' . $this->getTitle());
?>
<!DOCTYPE html>
<!--[if lt IE 7 ]> <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie6"> <![endif]-->
<!--[if IE 7 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie7"> <![endif]-->
<!--[if IE 8 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie8"> <![endif]-->
<!--[if IE 9 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie9"> <![endif]-->
<!--[if (gt IE 9)|!(IE)]><!--> <html dir="<?php echo $this->direction;
 ?>" lang="<?php echo  $this->language; ?>" class="<?php echo $b . ' '
 . $b . $v; ?>"> <!--<![endif]-->
```

The first line prevents unauthorized people from looking at your coding and potentially causing trouble. Then we grab a reference to the global site configuration. Next, we push some scripts to the document, first checking if the jquery plugin is enabled. Following that, we get the current site visitors browser and browser version. We add this to the document root as classes for better targeting with CSS. The last line of PHP takes the current page title and prepends the site's name. Thus, every page results with a title like "myHUB.org - My Page Title".

The first line of actual HTML tells the browser (and webbots) what sort of page it is. The next line says what language the site is in.

```
<head>
 <link rel="stylesheet" type="text/css" media="screen" href="<?php
 echo HubzeroDocumentAssets::getSystemStylesheet(array(
  'fontcons', 'reset', 'columns', 'notifications', 'pagination',
  'tabs', 'tags', 'comments', 'voting', 'layout'
 )); /* reset MUST come before all others except fontcons */ ?>" />
 <!-- Include the template's main CSS file -->
 <link rel="stylesheet" type="text/css" media="screen" href="<?php ech
o $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/main.
css" />
 <link rel="stylesheet" type="text/css" media="print" href="<?php echo
 $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/print.
```

```
css" />

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

 <!--[if IE 9]>
  <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/brow
ser/ie9.css" />
 <![endif]-->
 <!--[if IE 8]>
  <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/brow
ser/ie8.css" />
 <![endif]-->
 <!--[if IE 7]>
  <link rel="stylesheet" type="text/css" media="screen" href="<?php ec
ho $this->baseurl ?>/templates/<?php echo $this->template; ?>/css/brow
ser/ie7.css" />
 <![endif]-->
</head>
```

The first line compiles several bootstrap CSS files into a single, minified (comments and white-space removed to lessen file size) file to reduce http requests.

The following two lines include the main stylesheet for the template and a print stylesheet that applies more suitable styles when printing.

The fifth line gets Joomla! to put the correct header information in. This includes the page title, meta information, your main.css, system JavaScript, as well as any CSS or JavaScript that was pushed to the template from an extension (component, module, or plugin). This is a bit different than Joomla! 1.5's typical behavior in that the HUBzero code is automatically finding and including main.css and some key JavaScript files from your template. This is done due to the fact that order of inclusion is important for both CSS and JavaScript. For instance, one cannot execute JavaScript code built using the MooTools framework *before* the framework has been included. It would simply fail. As such, the naming and existence of specific directories, CSS, and JavaScript files becomes quite important for a HUBzero template.

The rest creates links to a couple CSS fix style sheets for Internet Explorer (more on this in the Cascading Style Sheets chapter).

Now for the main body:

```
<body>

 <div id="header">
  <h1><a href="<?php echo $this->baseurl ?>" title="<?php echo Config:
:get('sitename'); ?>"><?php echo Config::get('sitename'); ?></a></h1>

  <ul id="toolbar" class="<?php if (!$juser->get('guest')) { echo 'log
gedin'; } else { echo 'loggedout'; } ?>">
<?php
 // Is the user logged in?
 if (!User::isGuest()) {
  // Yes. Show them a different toolbar.
  echo '<li id="logout"><a href="/logout"><span>'.Lang::txt('Logout').
'</span></a></li>';
  echo '<li id="myaccount"><a href="/members/'.User::get('id').'"><spa
n>'.Lang::txt('My Account').'</span></a></li>';
  echo '<li id="usersname">'.User::get('name').' ('.User::get('usernam
e').')</li>';
 } else {
  // No. Show them the login and register options.
  echo "ttt".'<li id="login"><a href="/login" title="'.Lang::txt('Logi
n').'">'.Lang::txt('Login').'</a></li>'."n";
  echo "ttt".'<li id="register"><a href="/register" title="'.Lang::txt
('Sign up for a free account').'">'.Lang::txt('Register').'</a></li>'.
"n";
 }
?>
  </ul>

  <!-- Include any modules for the "search" position -->
  <jdoc:include type="modules" name="search" />
 </div><!-- / #header -->

 <!-- Include any modules assigned to the "user3" position -->
 <div id="nav">
  <h2>Navigation</h2>
  <jdoc:include type="modules" name="user3" />
 </div><!-- / #nav -->

 <div id="wrap">
  <div id="content" class="<?php echo $option; ?>">
   <!-- Include the component output -->
   <jdoc:include type="component" />
  </div><!-- / #content -->

  <div id="footer">
```

```
   <!-- Include any modules assigned to the "footer" position -->
   <jdoc:include type="modules" name="footer" />
  </div><!-- / #footer -->
 </div><!-- / #wrap -->
</body>
```

First we layout the site's masthead in the <div id="header"> block. Inside, we set the <h1> tag to the site's name, taken from the global site configuration.

Next, we move on to a toolbar that is present in the masthead of every page. This toolbar contains "login" and "register" links when not logged in and "logout" and "My Account" links when logged in. While not required, it is highly recommended that all templates include some form of this arrangement in an easy-to-find, consistent location.

Some modules that have been assigned the position "search" are then loaded in the masthead. Most HUBzero templates default to having a simple search form module appear. Again, this is not required and placement of modules is entirely up to the developer(s) but we, once again, strongly recommend that some form of a search box be included on all pages.

Then we move on to a block where navigation is loaded. It is here that our main menu will appear.

Next, we get to the primary content block. One of the first things you may notice is the use of module as a jdoc:include type. This is how we tell where in our template to output modules that have been assigned to specific positions.

It is also worth noting the small bit of PHP (<?php echo $option; ?>) in the class attribute of the content <div>. This small bit of code outputs the name of the current component as a CSS class. So, if one were on a page of a "groups" component, the resulting HTML would be <div id="content" class="com_groups">. Since all component output is contained inside the "content" div, this allows for more specific CSS targeting.

See the Modules: Loading article for more details on module positioning.

The content div contains a very important jdoc:include of type component. This is where all component output will be injected in the template. It is essential this line be included in a template for it to be able to display any content.

A Breakdown of error.php

Starting at the top:

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

// Get the user's browser and browser version
// We add this to the document root as classes for better targeting wi
th CSS
$browser = new HubzeroBrowserDetector();
$b = $browser->name();
$v = $browser->major();
?>
<!DOCTYPE html>
<!--[if lt IE 7 ]> &lthtml dir="&lt?php echo  $this->direction; ?>" la
ng="<?php echo  $this->language; ?>" class="ie6"> <![endif]-->
<!--[if IE 7 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie7"> <![endif]-->
<!--[if IE 8 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie8"> <![endif]-->
<!--[if IE 9 ]>    <html dir="<?php echo  $this->direction; ?>" lang="
<?php echo  $this->language; ?>" class="ie9"> <![endif]-->
<!--[if (gt IE 9)|!(IE)]><!--> <html dir="<?php echo $this->direction;
 ?>" lang="<?php echo  $this->language; ?>" class="<?php echo $b . ' '
 . $b . $v; ?>"> <!--<![endif]-->
```

The first line prevents unauthorized people from looking at your coding and potentially causing trouble. Then we grab a reference to the global site configuration. The first line of actual HTML tells the browser (and webbots) what sort of page it is. The next line says what language the site is in.

```
<head>
 <meta http-equiv="content-type" content="text/html; charset=utf-8" />
 <title><?php echo Config::get('sitename'); ?> - <?php echo $this->tit
le; ?> - <?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::get('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 Lang::txt('You may not be able to visit this page be
cause of:'); ?></p>

    <ol>
     <li><?php echo Lang::txt('An out-of-
date bookmark/favourite'); ?></li>
     <li><?php echo Lang::txt('A search engine that has an out-of-
date listing for this site'); ?></li>
     <li><?php echo Lang::txt('A mis-typed address'); ?></li>
     <li><?php echo Lang::txt('You have no access to this page'); ?></
li>
     <li><?php echo Lang::txt('The requested resource was not found');
 ?></li>
     <li><?php echo Lang::txt('An error has occurred while processing
your request.'); ?></li>
    </ol>

    <p><?php echo Lang::txt('If difficulties persist, please contact t
he system administrator of this site.'); ?></p>
   </div><!-- / #errorbox -->

   <form method="get" action="/search">
    <fieldset>
     <?php echo Lang::txt('Please try the'); ?> <a href="index.php" ti
tle="<?php echo Lang::txt('Go to the home page'); ?>"><?php echo Lang:
:txt('Home Page'); ?></a> <span><?php echo Lang::txt('or'); ?></span>
     <label>
      <?php echo Lang::txt('Search:'); ?>
      <input type="text" name="searchword" value="" />
     </label>
     <input type="submit" value="<?php echo Lang::txt('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 the CMS afterwards).

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

See the Modules: Loading article for further details on how to use more advanced features.

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.

Elements & Typography

Grid (Columns)

For laying out content on a page, the core hub framework includes styles for a 12-column grid.

...

...

...

...

...

...

...

...

...

...

...

...

The grid supports up to 12 columns with span# and offset# classes.

Each column **must** have a .col class. The last column in a set must have the .omega class added for IE 7 to work properly. No clearing div is required.

For example, a four column grid would look like:

```
<div class="grid">
 <div class="col span3">
  ...
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

...

Spanning Columns

Columns can be spanned to easier portion content on the page. In the following example, we span the first 6 columns in a container, then follow with two, smaller 3 column containers for a 3-column layout where the first column takes up 50% of the space.

```
<div class="grid">
 <div class="col span6">
  ...
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

Offsets

Columns may also be offset or 'pushed' over.

```
<div class="grid">
 <div class="col span3 offset3">
  ...
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

Helper Classes

.span-quarter	Span 3 columns. This is equivalent to .span3
.span-third	Span 4 columns. This is equivalent to .span4
.span-half	Span 6 columns. This is equivalent to .span6
.span-two-thirds	Span 8 columns. This is equivalent to .span8

.span-three-quarters
    Span 9 columns. This is equivalent to .span9

A four column grid with the helper classes:

```
<div class="grid">
 <div class="col span-quarter">
  ...
 </div>
 <div class="col span-quarter">
  ...
 </div>
 <div class="col span-quarter">
  ...
 </div>
 <div class="col span-quarter omega">
  ...
 </div>
</div>
```

There are equivalent .offset- classes as well:

.offset-quarter
    Offset 3 columns. This is equivalent to .offset3
.offset-third
    Offset 4 columns. This is equivalent to .offset4
.offset-half
    Offset 6 columns. This is equivalent to .offset6
.offset-two-thirds
    Offset 8 columns. This is equivalent to .offset8
.offset-three-quarters
    Offset 9 columns. This is equivalent to .offset9

Markup for a four column grid with the offset helper class:

```
<div class="grid">
 <div class="col span-quarter">
  ...
 </div>
 <div class="col offset-quarter span-quarter">
  ...
 </div>
 <div class="col span-quarter omega">
```

```
  ...
 </div>
</div>
```

Output:

...

...

...

Nesting Grids

The following is an example of a 3 column grid nested inside the first column of *another* 3 column grid.

```
<div class="grid">
 <div class="col span6">
  <div class="grid">
   <div class="col span4">
    ...
   </div>
   <div class="col span4">
    ...
   </div>
   <div class="col span4 omega">
    ...
   </div>
  </div>
 </div>
 <div class="col span3">
  ...
 </div>
 <div class="col span3 omega">
  ...
 </div>
</div>
```

Output:

...

...

...

...

...

Notifications

The core framework provides some base styles for alter and notifications.

```
<p class="passed">Success message</p>
```

Success message

```
<p class="info">Info message</p>
```

Info message

```
<p class="help">Help message</p>
```

Help message

```
<p class="warning">Warning message</p>
```

Warning message

```
<p class="error">Error message</p>
```

Error message

Sections & Asides

The majority of hub components have content laid out in a primary content column with secondary navigation or metadata in a smaller side column to the right. This is done by first wrapping the entire content in a div with a class of .section. The content intended for the side column is wrapped in a <div class="aside"> tag. The primary content is wrapped in a <div class="subject"> tag and immediately follows the .aside column.

Note: The .aside column must come first in order for the content to be positioned properly. If, unfortunately, this poses a semantic problem, we recommend using the grid system as a potential alternative.

Using aside & subject differs from the grid system in that the .aside column has a fixed width with the .subject column taking up the available left-over space. In the grid system, **every** column is flexible (uses a percentage of the screen) and cannot have a specified, fixed width.

Example usage:

```
<section class="section">
 <div class="section-inner">
  <div class="aside">
   Side column content ...
  </div>
  <div class="subject">
   Primary content ...
  </div>
 </div>
</section>
```

Buttons

{xhub:include type="stylesheet" filename="/media/system/css/buttons.css"}

States

default disabled active

```
<a class="btn" href="#">default</a>

<a class="btn disabled" href="#">disabled</a>

<a class="btn active" href="#">active</a>
```

Size

primary secondary

```
<a class="btn btn-primary" href="#">primary</a>

<a class="btn btn-secondary" href="#">secondary</a>
```

Type

link button

```
<a class="btn" href="#">link</a>

<button class="btn" href="#">button</button>

<input type="submit" class="btn" value="input" />
```

Color

danger warning info success

```
<a class="btn btn-danger" href="#">danger</a>

<a class="btn btn-warning" href="#">warning</a>

<a class="btn btn-info" href="#">info</a>

<a class="btn btn-success" href="#">success</a>
```

Icons

danger

warning

info

success

edit

delete

delete

secondary

```
<a class="btn btn-danger icon-danger" href="#">danger</a>

<a class="btn btn-warning icon-warning" href="#">warning</a>

...
```

Groups

Dropdown

- Action
- Another action
- Something else here
-
- Separated link

```
<div class="btn-group dropdown">
          <a class="btn" href="#">Dropdown</a>
          <span class="btn dropdown-toggle"></span>
          <ul class="dropdown-menu">
            <li><a href="#">Action</a></li>
            <li><a href="#">Another action</a></li>
            <li><a href="#">Something else here</a></li>
            <li class="divider"></li>
            <li><a href="#">Separated link</a></li>
          </ul>
</div>
```

Dropup

- Action
- Another action
- Something else here
-
- Separated link

```
<div class="btn-group dropup">
          ...
</div>
```

Dropdown

- Action
- Another action
- Something else here
-
- Separated link

```
<div class="btn-group btn-secondary dropdown">
          ...
</div>
```

prev all next

```
<div class="btn-group">
          <a class="btn icon-prev" href="#">prev</a>
          <a class="btn" href="#">all</a>
          <a class="btn icon-next opposite" href="#">next</a>
</div>
```

Languages

Overview

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

```
/templates
.. /foo
.. .. /languages
.. .. .. /{LanguageName}
.. .. .. .. {LanguageName}.tpl_{TemplateName}.ini
```

Setup

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

```
; Template Test (en-US)
TPL_TEST_HERE_IS_LINE_ONE = "Here is line one"
TPL_TEST_HERE_IS_LINE_TWO = "Here is line two"
TPL_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 TPL_{TemplateName}_{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 template is preloaded, as long as it follows the naming conventions detailed above, when the template is rendered. As such, no manual loading is necessary. However, if you wish to load an alternate language file, you can do so by calling

Lang::load($extension);. The following example demonstrates, from a template layout, loading a language file for the component 'com_test' out of the /hub/app/languages directory.

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

Lang::load('com_test', PATH_APP . '/languages');
?>
<html>
...
```

Translating Text

Below is an example of accessing the translate helper:

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

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

See the Languages overview for details.

Migrations

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

To illustrate the typical component directory structures and files:

```
/app
.. /templates
.. .. /{TemplateName}
.. .. .. /css
.. .. .. /html
.. .. .. /img
.. .. ..  /js
.. .. .. /migrations
.. .. .. .. /Migration20190301102219TplExample.php
.. .. .. error.php
.. .. .. component.php
.. .. .. index.php
.. .. .. templateDetails.xml
.. .. .. template_thumbnail.png
.. .. .. favicon.ico
```

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

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

```
<?php

use HubzeroContentMigrationBase;

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

/**
 * Migration script for registering the example plugin
 **/
class Migration20190301102219TplExample extends Base
{
    /**
```

```
     * Up
     **/
    public function up()
    {
        // Register the template
        //
        // @param   string  $element    Template element
        // @param   string  $name       (option) Template name
 // @param   int     $client     (optional, default: 1) Admin (1) or s
ite (0) client
 // @param   int     $enabled    (optional, default: 1) Whether or not
 the template should be enabled (1=yes, 0=no)
 // @param   int     $home       (optional, default: 0) Whether or not
 this should become the enabled/home template (1=yes, 0=no)
 // @param   array   $styles     (optional) Template styles
        $this->addTemplateEntry('example', 'example', 0, 1, 0);
    }

    /**
     * Down
     **/
    public function down()
    {
        // Unregister the template
        //
 // @param   string  $name    Template element name
 // @param   int     $client  Client id
        $this->deleteTemplateEntry('example', 0);
    }
}
```

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