6
Design Templates – Part Two

In the previous chapter we built the basics of templating, including how to use Smarty, and how to set up a theme so the CMS can display pages through the templates.

We also built a basic HTML navigation menu.

In this chapter, we will: Finish the templating engine. Improve the navigation menu using the Filament Group menu.

At the end of this chapter, the CMS will be complete enough to use in simple sites.

Adding jQuery to the menu

For a very long time, I was using a home-grown JavaScript navigational menu.

It was capable of displaying in many different ways—drop downs, slide downs (like jQuery-UI accordions), fade-ins.

It had built-in collision checks to make sure it was always visible and didn't try to render past the sides, bottom, or top of a screen.

It was complex, and I would only ever touch it when I was asked to add yet another feature to it by a client, or when a bug was discovered.

That's never the ideal situation. In an ideal situation, the components you use in your system are constantly being improved and added to, even when you are not working on it yourself.

That's where open source comes into its own. I really do love using a piece of software for a few months, then finding it has been updated by the developers and now has a load of new features that I wasn't aware that I wanted, but now "need".

In my CMS, I've replaced my home-grown solution with an existing project by the Filament Group, which can be seen in action here: `http://www.filamentgroup.com/lab/jquery_ipod_style_and_flyout_menus/`.

If you read through that document, you will see that the group is no longer working on the project, because they've given it to the jQuery-UI team to help create a jQuery-UI menu plugin.

At the time of writing, the jQuery-UI menu is still in development, and should be available by version 1.9. It's currently in a very basic state and unusable, but whenever the jQuery-UI team focuses on anything, the end result is always comprehensive and amazingly stable.

In the meantime, the existing **Filament Group Menu** (**fg-menu** from now on) is probably the best "general use" menu out there, and I'm certain that when the jQuery-UI version is released, porting from the fg-menu system to the new plugin will be easy.

By "general use", I mean that it is not designed specifically to be a drop down or fly-out menu. It's not designed to look exactly one way, or work exactly one way. It can work in a few different ways, so the site designers are not constrained too much by what we developers force on them.

So, let's install it.

Preparing the Filament Group Menu

Download the fg-menu code from the previously mentioned page (search for **Download the script, CSS, and sample HTML** on the page) and unzip it in `/j`. A `/j/__MACOSX` directory and a `/j/fg-menu` directory will both be created. Delete the `/j/__MACOSX` directory.

One problem with downloading plugins is that sometimes, the writers will associate colors and other styles to the elements that you will need to overwrite.

In most cases, I'd advocate adding a CSS sheet which overrides the fg-menu by using more specific selectors. This has a disadvantage of having the browser download two sheets when only one is needed.

However, since we know that the current version of the plugin is the last ever until jQuery-UI 1.9 is released, I feel it is okay to edit the downloaded files themselves.

This includes the JavaScript files. There are a number of things I didn't like about the fg-menu JavaScript, and the easiest way to address them was by editing the source itself.

I won't describe the CSS changes here (they're in the downloadable code bundle available on Packt's website) other than to say it was purely to remove colors.

The JavaScript gripes are minor as well, but they were enough that I felt the need to hack the source.

The default code forces the user to click the menu to activate its sub-menus.

This has the disadvantage that if you have say two pages, "Page1" and "Page1>Page2", where Page2 is a sub-page of Page1, then how do you tell the menu that you want to go to Page1? Clicking should do it, but instead, it opens the sub-menu!

The solution for this is easy: Just replace the responsible `.click()` events with `.mouseover()` events (lines 26 and 363) in the file `/j/fg-menu/fg.menu.js`.

Another problem has to do with widths and heights. I only noticed this on IE (no other browser) with jQuery 1.4.

fg-menu has two custom functions, `jQuery.fn.getTotalWidth()` and `jQuery.fn.getTotalHeight()` (lines 549 to 556), but those are no longer necessary, because you can use jQuery's `.outerWidth()` and `.outerHeight()` functions.

So, delete the source for those two custom functions, and edit lines 468 and 469 to refer to `outerWidth()` and `outerHeight()` instead.

I'm walking through the process that I used to fix the code to try to explain how it was done. If you prefer to skip copying the process yourself, you can download the finished code as part of this chapter's downloadable code bundle from the Packt website.

Another thing is the `this.chooseItem()` function. In fg-menu, this is called when an item is clicked. In our case, we always want this to actually go to the page that's clicked. So add this line to the beginning of the function (after line 244):

```
location.href = $(item).attr('href');return;
```

I've placed both commands on the same line because I want to make as few changes to the original structure as possible, so that if I ever need to refer to a line number, it's as close to the original source as possible.

There are some other minor issues, but not important enough to mention here (they're all fixed in the chapter's code bundle).

The final change I made to the plugin is something that was actually requested of the Filament Group by an interested user, but they'd passed on responsibility by that time and it was never answered.

When a menu is opened and you take the mouse off it, it is expected that the menu will close. This doesn't happen in fg-menu (you need to actually click the document to close it).

To fix this, I added the following jQuery code to the end of the file:

```
$('.fg-menu,.fg-menu-top-level')
  .live('mouseover',function(){
    this.mouse_is_over=true;
    clearTimeout(window.fgmenu_mouseout_timer);
  })
  .live('mouseout',function(){
    this.mouse_is_over=false;
    window.fgmenu_mouseout_timer=setTimeout(function(){
      var o=0;
      $('.fg-menu,.fg-menu-top-level').each(function(){
        if (this.mouse_is_over) o++;
      });
      if(!o){
        $.each(allUIMenus, function(i){
          if (allUIMenus[i].menuOpen) {
            allUIMenus[i].kill();
          };
        });
      }
    },2000);
  });
```

In short, this code tells `fg-menu` to close all menus two seconds after the mouse has left it.

Let's examine this in more detail.

When you move your mouse between two elements that appear to be right next to each other, it is possible that the browser will interpret even the slightest gap (border, margin, and so on) as meaning the mouse is not in either of them.

For this reason, we need to create a "grace" period, which allows the mouse time to move from one to the other.

How we do that is when the mouse enters a menu item, you set a variable `mouse_is_over` on it. When the mouse leaves the item, we unset that variable, and start a countdown to the destruction code.

The countdown (a two second `setTimeout`) gives us enough time to move the mouse to another item and disable the timer.

If by chance the timer still goes off, for example the menu you left overlaps or is contained in another menu, and the `mouseenter` event never triggered on the "container", then the destruction code does a test to see if `mouse_is_over` is set.

If so, it does nothing. If not, then all menu entries are killed.

Integrating the menu

We've already embedded a `<ul>` tree version of the menu where `{{MENU}}` appears in the template. Enhancing this involves simply applying the fg-menu plugin to that `<ul>`.

Using the plugin involves adding the source of the plugin as an external JavaScript reference.

If we simply echo out a `<script>` tag every time we need to reference a file, we may end up with redundant loads.

For example, let's say you had two menus on the page. It is silly to have to load a static script twice (once for each menu), so let's create a global array which holds a list of external scripts and CSS files that need to be loaded, and whenever we want to output a script, we'll check against it.

Edit `/index.php` and add the following highlighted lines:

```
// { common variables and functions
include_once('ww.incs/common.php');
$page=isset($_REQUEST['page'])?$_REQUEST['page']:'';
$id=isset($_REQUEST['id'])?(int)$_REQUEST['id']:0;
$external_scripts=array();
$external_css=array();
// }
```

And we will add these functions to `/ww.incs/common.php`:

```
function import_script_once($script){
  global $external_scripts;
  if(isset($external_scripts[$script]))return '';
  $external_scripts[$script]=1;
  return '<script src="'.htmlspecialchars($script).'">
      </script>';
}
function import_css_once($css){
  global $external_css;
  if(isset($external_css[$css]))return '';
  $external_css[$css]=1;
  return '<link rel="stylesheet" href="'
      .htmlspecialchars($css).'"/ >';
}
```

It is easier to ensure that `$array[$filename]` is unique (as an array key), than to ensure that `$filename` is unique in `$array[]` as a value.

Now let's add the fg-menu script references. Edit `/ww.incs/common.php` and add these highlighted lines to the top of the `fg_menu_show()` function:

```
function menu_show_fg($opts){
  $c='';
  $c.=import_script_once('/j/fg-menu/fg.menu.js');
  $c.=import_css_once('/j/fg-menu/fg.menu.css');
  $options=array(
```

And now we add the code to activate the conversion from `<ul>` tree to `flyOut` menu. Add this to the end of the same function in the same file. I've highlighted the lines that the code goes between:

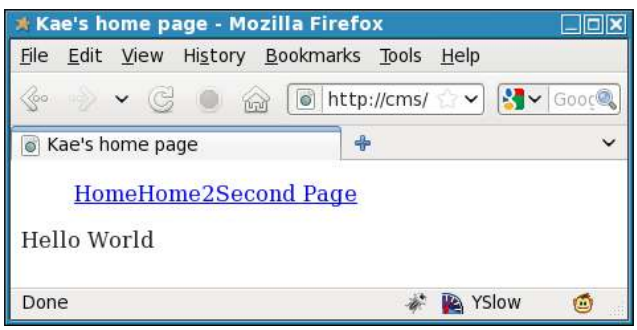
```
  $c.='<div class="menu-fg menu-fg-'.$options['direction']
      .'" id="menu-fg-'.$menuid.'">'
      .menu_build_fg($options['parent'],0,$options).'</div>';
  if($options['direction']=='vertical'){
    $posopts="positionOpts: { posX: 'left', posY: 'top',
      offsetX: 40, offsetY: 10, directionH: 'right',
      directionV: 'down', detectH: true, detectV: true,
      linkToFront: false },";
  }
  else{
    $posopts='';
  }
  $c.="<script>
jQuery.fn.outer = function() {
  return $( $('<div></div>').html(this.clone()) ).html();
}
$(function(){
  $('#menu-fg-$menuid>ul>li>a').each(function(){
    if(!$(this).next().length)return; // empty
    $(this).menu({
      content:$(this).next().outer(),
      choose:function(ev,ui){
        document.location=ui.item[0].childNodes(0).href;
      },
      $posopts
      flyOut:true
    });
  });
  $('.menu-fg>ul>li').addClass('fg-menu-top-level');
});
</script>";
  return $c;
}
```

First off, we set up the `$posopts` variable, which will tell fg-menu where sub-menus should be positioned relative to their parents. For example, where the top-level menu is vertical, the sub-menu should be offset to the right-hand side. In horizontal top-level menus, the sub-menu should appear directly below its parent (the default).

Next, we output the JavaScript which sets up the menu.

Notice that we've added a short jQuery plugin inline—because most browsers don't provide a method to get the outer HTML of an element, and fg-menu generates its sub-menus from an HTML string, we need to add this function so we can generate the HTML from the `<ul>` tree.

Now when you reload the browser, the screen will look like this, with the sub-menus hidden:



Because we are using an absolutely basic template, there is no margin or padding set up (in fact, the theme's CSS sheet is still blank at this point of the chapter). Remember that we set up the template initially to have the menu embedded with its direction set to "horizontal".

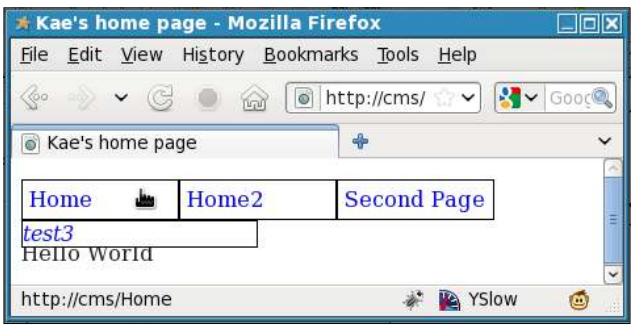
Embedding the `{{MENU}}` template function will automatically set up a basic menu. You need to then add your own CSS to make it look better. As an example, here is some simple CSS which I've added to the example template's CSS file, `/ww.skins/basic/c/style.css`:

```
.fg-menu-container a{
  border:1px solid #000;
  text-decoration:none;
  font-style:italic;
  background:#fff;
}
.menu-fg a{
  border:1px solid #000;
  padding:5px;
```

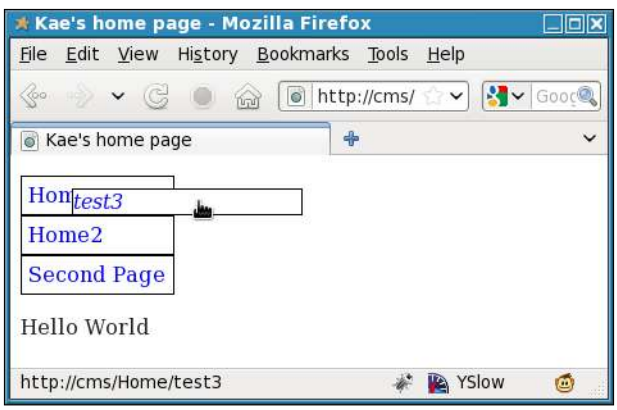
```
    text-decoration:none
}
.menu-fg li{
    width:120px;
}
.menu-fg ul{
    list-style:none;
    padding:0;
}
```

That's enough to make the menu more usable.

Here's a screenshot with the first item opened:



And if we edit the template and replace the "horizontal" with "vertical", we get this:



That's it for now with the menus. We'll come back to them later on when we're working on a menu plugin (so administrators can add menus to template without needing to actually edit the template source).

Until then, if you only plan on having one theme in your CMS, with only one template, there is enough of the engine built now for you to create simple sites.

However, if you want to provide multiple themes, we will need to build that into the administration area of the CMS. Let's do that now.

Choosing a theme in the administration area

Okay—let's write the theme switcher.

First, we will add the `Themes` page to the admin menu. The menu is getting a bit full, but we'll take care of that in the next chapter when we consider plugins.

Edit `/ww.admin/header.php` and add the highlighted line:

```
<li><a href="/ww.admin/users.php">Users</a></li>
<li><a href="/ww.admin/themes.php">Themes</a></li>
<li>
  <a href="/ww.incs/logout.php?redirect=/ww.admin/">
    Log Out
  </a>
</li>
```

Eventually, we will want to group the site-management (versus page-management) functions together, so we will add this to the `Users` admin page menu as well. Edit `/ww.admin/users.php` and add this highlighted line:

```
echo '<a href="/ww.admin/users.php">Users</a>';
echo '<a href="/ww.admin/themes.php">Themes</a>';
echo '</div>';
```

Similar to the Users page, we will have a "wrapper" file in `/ww.admin` that loads up sub-requirements. You can create this file by copying `/ww.admin/users.php`, making the small changes necessary (highlighted) to make it themes-based, and save it as `/ww.admin/themes.php`:

```
<?php
require 'header.php';
echo '<h1>Theme Management</h1>';

echo '<div class="left-menu">';
echo '<a href="/ww.admin/users.php">Users</a>';
echo '<a href="/ww.admin/themes.php">Themes</a>';
echo '</div>';

echo '<div class="has-left-menu">';
echo '<h2>Theme Management</h2>';
```

```
require 'themes/list.php';
echo '</div>';

echo '<script src="/ww.admin/themes/themes.js"></script>';
require 'footer.php';
```

Now create the directory `/ww.admin/themes`. We will place the dependent files in there.

Anything up to twenty or thirty themes can be easily displayed on one page to be chosen from by the administrator.

If there are more, then a more advanced selection script will need to be created than the one described in this section.

It is unanticipated, though, that in a single-site CMS, any more than two or three would be added to the repository at any time—after all, people don't switch their designs every two weeks!

In a larger system, though, where the CMS may be one of many instances which are accessing a common repository (in the case of a large hosting company that offers off-the-shelf websites, for example), there could be hundreds or even possibly thousands.

So, we've already created one simple theme, called "Basic", which really could not be any simpler.

However, when offering it up for selection (along with others), the name of the design is not really enough—it is better to show a screenshot so the admin has a visual idea of what they are choosing.

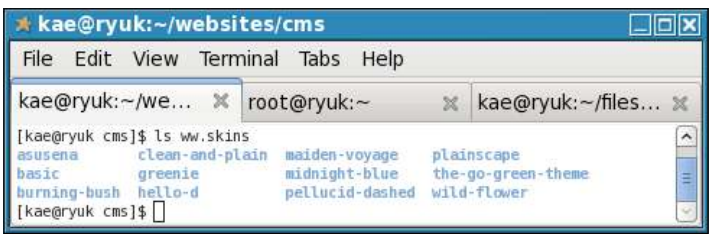
In larger systems, you may also have a description, describing the basic colors, whether the theme has columns, requires certain plugins, and so on. We will not need these.

So, first, load up your site, and take a screenshot of the design. Save that design as `/ww.skins/basic/screenshot.png`.

This is the convention that we will use—inside each theme directory, there will be a `screenshot.png`, sized `240x172` pixels. If there is no such file, then it will not be displayed in the admin area. As a side benefit, this will also allow you to "deprecate" any old designs, by hiding them from the admin, yet still allowing the design to work if it is already selected.

To demonstrate this sub-project, I've added eleven directories to my `/ww.skins` directory, with screenshots in each taken from freely-available WordPress designs (available here: `http://wordpress.org/extend/themes/browse/popular/`) — while WordPress themes will not work directly in the CMS, it is actually very simple to convert them so they do, either by hand or with a script.

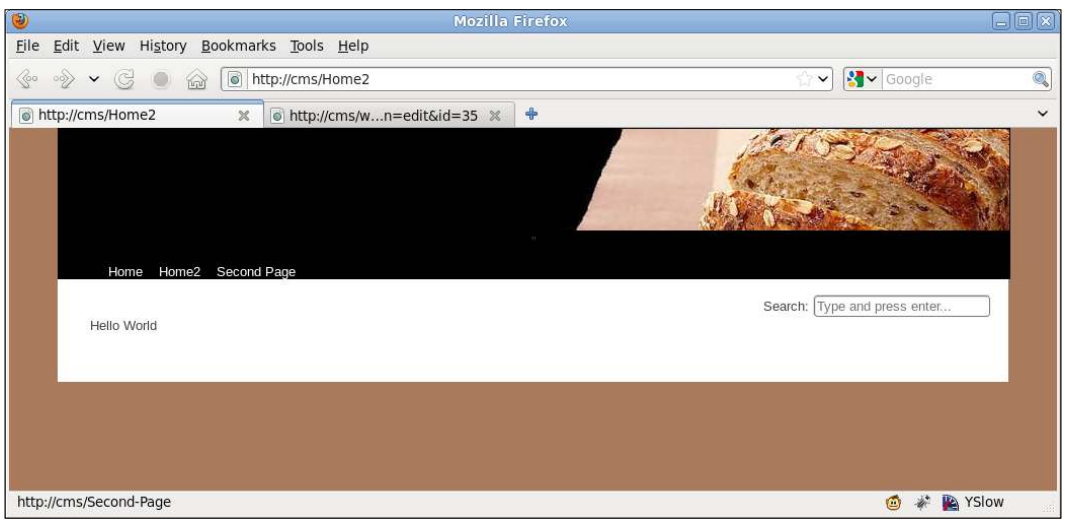
Here is my `/ww.skins` directory:



A shrewd reader will note that at the moment, the CMS does not currently save which theme it is using, and instead simply chooses the first it finds in that directory.

The order of files in a directory is not necessarily alphabetical.

When I load up my browser and check the front page again, I find it is no longer using the `Basic` theme, but the one named "Bakery":



So how do we get the CMS to store one theme and not randomly choose others?

Earlier in the chapter we wrote some code which checked to see if the theme was defined in the `/.private/config.php` file's `$DBVARS` array, and if not, then choose from the `/ww.skins` directory. So we need to be able to change that array on-the-fly from the admin area.

The way to manage this is to make the file writable by the web server, as described back in the KFM section of *Chapter 3*, *Page Management – Part One*, then add this function to `/ww.incs/basics.php`:

```
function config_rewrite(){
  global $DBVARS;
  $tmparr=$DBVARS;
  $tmparr2=array();
  foreach($tmparr as $name=>$val)$tmparr2[]=
      '\''.addslashes($name).'\'=>\''.addslashes($val).'\'';
  $config="<?php\n\$DBVARS=array(\n "
      .join(",\n ",$tmparr2)
      ."\n);";
  file_put_contents(CONFIG_FILE,$config);
}
```

What this does is to take the current global `$DBVARS` array, and re-create it as an executable PHP string, and write it back into `CONFIG_FILE` (`/.private/config.php` by default).

Now in order to set the theme, all we need to do is to add a `theme` field to the global `$DBVARS` array and then call `config_rewrite()`.

There is one more function needed. Add this to the same file:

```
function cache_clear($type){
  if(!is_dir(SCRIPTBASE.'/ww.cache/'.$type))return;
  $d=new DirectoryIterator(SCRIPTBASE.'/ww.cache/'.$type);
  foreach($d as $f){
    $f=$f->getFilename();
    if($f=='.' || $f=='..')continue;
    unlink(SCRIPTBASE.'/ww.cache/'.$type.'/'.$f);
  }
}
```

The reason for this is that Smarty caches the templates based on the filename in the template directory. But, because each template contains the same filenames, Smarty gets confused and reuses the old cache.

To solve this, we clear the cache whenever a new theme is chosen. We'll talk more about caches later on in the chapter.

Now let's create `/ww.admin/themes/list.php`:

```
<?php
// { handle actions
if(isset($_REQUEST['action']) && $_REQUEST['action']=='set_theme'){
  if(is_dir(THEME_DIR.'/'.$_REQUEST['theme'])){
    $DBVARS['theme']=$_REQUEST['theme'];
    config_rewrite();
    cache_clear('pages');
  }
}
// }
// { display list of themes
  $dir=new DirectoryIterator(THEME_DIR);
  $themes_found=0;
  foreach($dir as $file){
    if($file->isDot())continue;
    if(!file_exists(THEME_DIR.'/'.$file.'/screenshot.png'))
        continue;
    $themes_found++;
    echo '<div style="width:250px;text-align:center;
        border:1px solid #000;margin:5px;height:250px;
        float:left;';
    if($file==$DBVARS['theme'])echo 'background:#ff0;';
    echo '"><form method="post" action="./themes.php">
        <input type="hidden" name="page" value="themes" />
        <input type="hidden" name="action"
            value="set_theme" />';
    echo '<input type="hidden" name="theme"
        value="'.htmlspecialchars($file).'" />';
    $size=getimagesize(
        '../ww.skins/'.$file.'/screenshot.png');
    $w=$size[0]; $h=$size[1];
    if($w>240){
      $w=$w*(240/$w);
      $h=$h*(240/$w);
    }
    if($h>172){
      $w=$w*(172/$h);
      $h=$h*(172/$h);
    }
    echo '<img src="/ww.skins/'.htmlspecialchars($file)
        .'/screenshot.png" width="'.(floor($w)).'"
        height="'.(floor($h)).'" /><br />';
```

```
        echo '<strong>',htmlspecialchars($file),'</strong><br />';
        echo '<input type="submit" value="set theme" />
            </form></div>';
      }
      if($themes_found==0){
        echo '<em>No themes found. Create a theme and place it
            into the /ww.skins/ directory.</em>';
      }
    // }
```

At the head of the file, we check to see if any action was requested (to see if a theme was chosen).

If so, we set that in `$DBVARS` and rewrite the config file, then clear the Smarty cache so the new template is used instead of the old cached one.

Next, we display a form for each theme in `/ww.skins` that has a `screenshot.png` file in it.

We display the screenshot, making sure to resize it down if it's larger than a certain size (I chose `240x172`), keeping the aspect ratio so it doesn't look weird.

With all this, you should now be able to click on **set theme** and have it update the configuration file. Make sure of this by checking `/.private/config.php` after clicking, to see if you got the file permissions right.

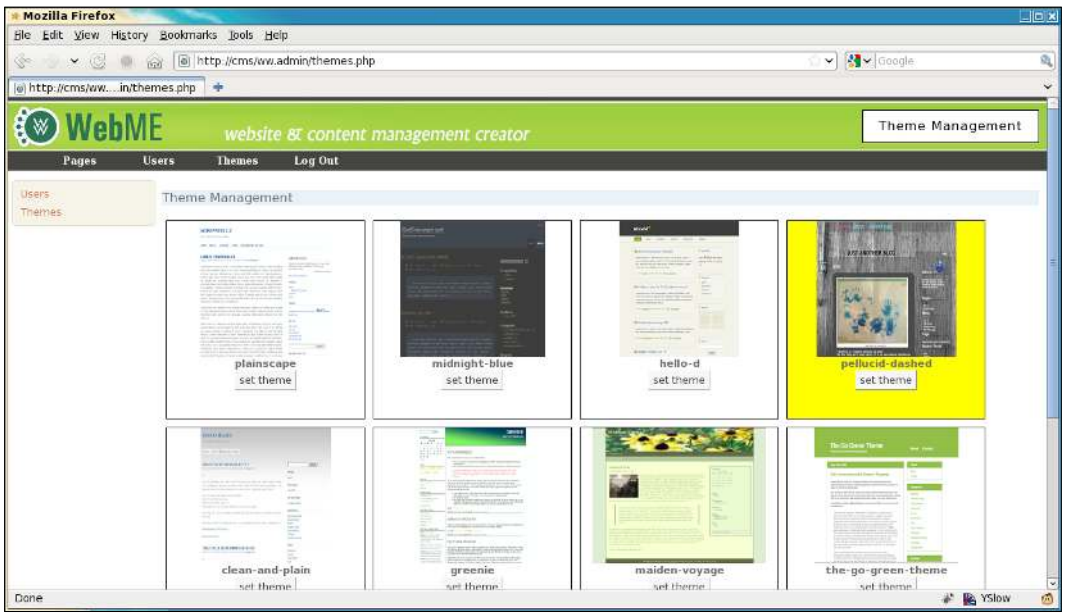
Here's an example before clicking:

```
<?php
$DBVARS=array(
  'username'=>'cmsuser',
  'password'=>'cmspass',
  'hostname'=>'localhost',
  'db_name'=>'cmsdb'
);
```

And after clicking, that file is updated to this, with the selected theme highlighted:

```
<?php
$DBVARS=array(
  'username'=>'cmsuser',
  'password'=>'cmspass',
  'hostname'=>'localhost',
  'db_name'=>'cmsdb',
  'theme'=>'basic'
);
```

Oh, and here is what the theme selection page looks like (with the **pellucid-dashed** theme selected—upper right-hand side corner):



Next, we will update the Basic theme to have multiple templates, and add the ability to choose those templates.

Choosing a page template in the administration area

In the Basic theme, we created just one template, which we named `/ww.skins/basic/h/_default.html`. Edit that file, and make sure the menu is back to `horizontal`.

Now let's create a second template, called `/ww.skins/basic/h/menu-on-left.html`:

```
<!doctype html>
<html>
  <head>
    {{$METADATA}}
    <link rel="stylesheet"
        href="/ww.skins/basic/c/style.css" />
  </head>
  <body class="menu-on-left">
    <div id="menu-wrapper">{{MENU direction="vertical"}}</div>
```

```
    <div id="page-wrapper">{{$PAGECONTENT}}</div>
  </body>
</html>
```

Notice the class `menu-on-left`. That lets us add the following to the CSS sheet at `/ww.skins/basic/c/style.css`:

```
.menu-on-left #menu-wrapper{
  float:left;
  width:130px;
}
.menu-on-left #page-wrapper{
  margin-left:140px;
}
```

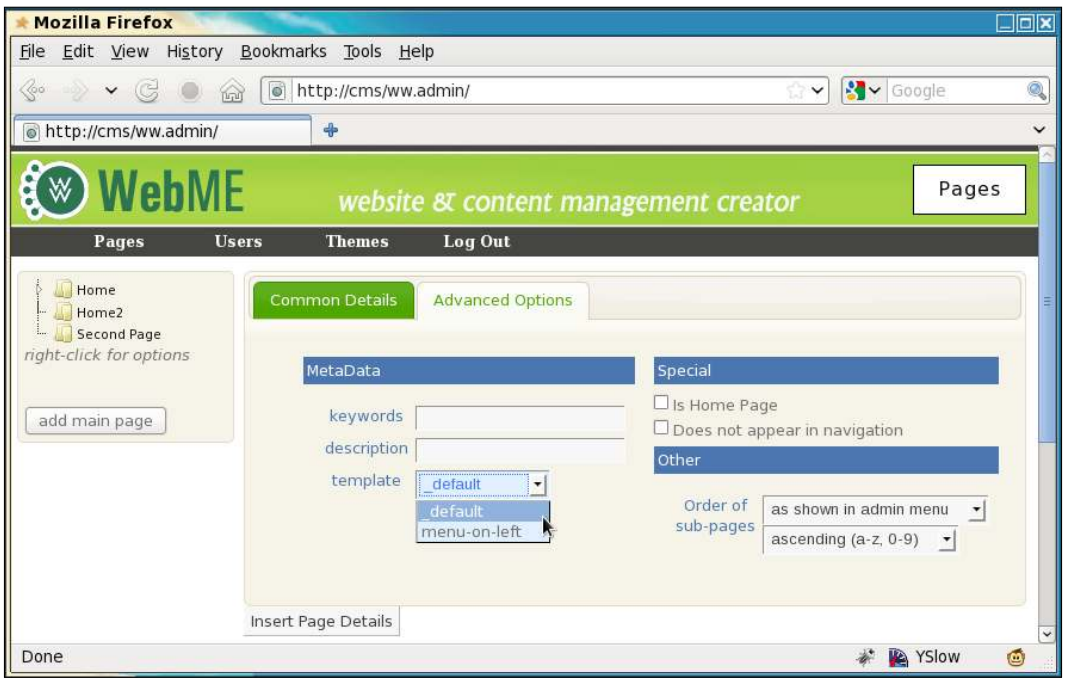
This will only affect the menu wrapper and page wrapper in that specific template.

Now open `/ww.admin/pages/forms.php` and where it says `we'll add this in the next chapter`, replace that block with this:

```
// { template
echo '<tr><th>template</th><td>';
$d=array();
if(!file_exists(THEME_DIR.'/'.THEME.'/h/')){
  echo 'SELECTED THEME DOES NOT EXIST<br />Please
      <a href="/ww.admin/siteoptions.php?page=themes">select
      a theme</a>';
}
else{
  $dir=new DirectoryIterator(THEME_DIR.'/'.THEME.'/h/');
  foreach($dir as $f){
    if($f->isDot())continue;
    $n=$f->getFilename();
    if(preg_match('/\.html$/',$n))
        $d[]=preg_replace('/\.html$/','',$n);
  }
  asort($d);
  if(count($d)>1){
    echo '<select name="template">';
    foreach($d as $name){
      echo '<option ';
      if($name==$page['template'])echo ' selected="selected"';
      echo '>',$name,'</option>';
    }
    echo '</select>';
  }
  else echo 'no options available';
}
```

```
    echo '</td></tr>';
    // }
```

Straightforward enough—this block first checks that the selected theme actually exists, and then displays template options if there are any, with the already-selected one selected (or the first on the list if none are already selected).



In the screenshot, you can see the list of templates. `_default` is at the top alphabetically.

Now, you need to edit `/ww.admin/pages/action.edit.php`, and change the `create SQL` block to this:

```
    // { create SQL
    $q='template="'.addslashes($_REQUEST['template']).'",
        edate=now(),type="'.addslashes($_REQUEST['type']).'",
        associated_date="'.addslashes($associated_date).'",
        keywords="'.addslashes($keywords).'",
        description="'.addslashes($description).'",
        name="'.addslashes($name).'",
        title="'.addslashes($title).'",
        body="'.addslashes($body).'",parent='.$pid.',
        special='.$special.',vars="'.addslashes($vars).'"';
    // }
```

And with that done, you can now switch templates for each page on the front-end.

Running Smarty on page content

Let's say you want to embed some templated stuff into the actual content of the page.

For example, let's say that we've already done the next few chapters and have build the "image transitions" plugin. You want to have a load of images fading into each other in the page you are writing.

To do that, you would either have to write the source code of the transition effect into the page itself, or embed just the code for it. `{{IMAGE_TRANSITION directory="img"}}` is much easier to write than a whole code block, so it makes sense to use Smarty to handle not just the wrapping template, but also the page content itself.

To do that, you need to have the page content saved in a file, as Smarty works on actual files, and not on database stuff.

Edit `/ww.php_classes/Page.php` and add this method to the `Page` class:

```
function render(){
  $smarty=smarty_setup('pages');
  $smarty->compile_dir=SCRIPTBASE . '/ww.cache/pages';
  if(!file_exists(SCRIPTBASE.'/ww.cache/pages/template_'
      .$this->id)){
    file_put_contents(SCRIPTBASE.'/ww.cache/pages/template_'
        .$this->id,$this->body);
  }
  return $smarty->fetch(SCRIPTBASE
      .'/ww.cache/pages/template_'.$this->id);
}
```

When called, this method first sets up Smarty, as we saw earlier in the chapter.

Then it checks to see if a copy of the page body exists as a file in `SCRIPTBASE .'/ww.cache/pages/'`. If not, then the file is created.

Then, Smarty is run against that file and the result is returned.

Now we need to make sure it is used. Edit `/index.php`, and in the `set up pagecontent` section, change the first case to this:

```
case '0': // { normal page
  $pagecontent=$PAGEDATA->render();
  break;
// }
```

And finally, to make sure that the Smarty cache is cleared every time a page is changed, we add this (highlighted line) to the end of `/ww.admin/pages/action.edit.php`:

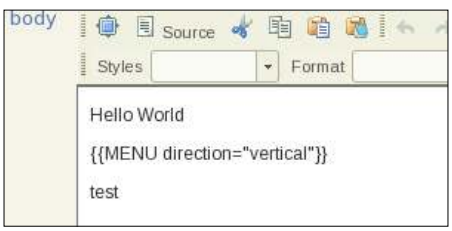
```
echo '<em>Page Saved</em>';
cache_clear('pages');
```

And also to the end of `/ww.admin/pages/delete.php`:

```
}
echo 1;
cache_clear('pages');
```

That will do it!

Now, as a test, change your home page to use the `_default` template (the one with the horizontal menu), and then edit the page body to this:



Before you save, there's something important to fix first. The CKeditor RTE (Rich-text Editor) converts the double-quote character `"` to the HTML entity code `&quot;` in the background. This can cause problems, so we will edit `/ww.admin/pages/action.edit.php`, and where the `$body` variable is initialised, convert that line to this highlighted one:

```
$name           =pages_setup_name($id,$pid);
$body           =str_replace('&quot;','"',$_REQUEST['body']);
$special        =pages_setup_specials($id);
```

Then click on **Update** to save the page.

Now when you view the front-end, you should see this:

And that demonstrates that you can now call Smarty functions from within the page body itself!

Summary

In this chapter, we advanced the CMS engine to the stage that you can now create a designed template, including page menus, and embed the page content within that.

Not only that, but you can also now use Smarty functions within the page body as well, meaning that when we create plugins, we can use some of them "inline" in the page body.

In the next chapter, we will start on the plugin framework, allowing us to add or remove modules of code without affecting the core code at all.