5
Design Templates – Part One

This chapter will demonstrate how the content of the website can be embedded in a design template.

The design of a site is also known as the "theme" or "skin". I will use these words interchangeably. They both refer to the same thing.

A theme is composed of one or more page templates. The page template is used to define a layout of a specific page (for example, a "splash page" vs. a content page), while the theme says overall what the website looks like—colors, images, and so on. Designers call this the "look and feel" of the website.

In this chapter, we will discuss:

- How themes and templates work
- Smarty templating engine
- Creation of a theme
- Front-end navigation

We will continue with these topics into the next chapter, which will improve on the navigation using jQuery, and then we'll build a theme management system.

How do themes and templates work?

A "theme" is a term which describes the overall look and feel of a website. It describes what the various elements look like—headers, links, tables, lists, and so on. It defines the colors that are used, and any common background images such as gradients or logos. The theme contains one or more templates, and any images or other resources that will be required by those templates.

A "template" is basically an HTML snippet which defines the layout of a page—where on the page the menu is located, are there panels, is there a header and footer. It uses the theme's design, so that other templates in the same site have a similar feel to them.

In the CMS we are building, a template uses a few codes to define where the various elements go on a page.

As an example, here is a very simple template, using code designed to work with the Smarty templating engine:

```
<!doctype html>
<html>
  <head>
    {{$METADATA}}
  </head>
  <body>
    {{MENU direction="horizontal"}}
    {{$PAGECONTENT}}
  </body>
</html>
```

The three highlighted lines are template codes which show where the CMS should place various HTML snippets that it generates.

PHP is itself described as a templating engine, as you can mix it in with HTML simply enough (and in fact, that's how it was originally designed).

You might ask, why bother using an external engine such as Smarty at all when PHP is one itself?

Here is the above template written as PHP:

```
<!doctype html>
<html>
  <head>
    <?php
    require 'common.php';
    echo $METADATA;
    ?>
  </head>
  <body>
    <?php
    MENU (array('direction'=>'horizontal'));
    echo $PAGECONTENT;
    ?>
  </body>
</html>
```

The difference here is not huge, but the first example is easier for a non-PHP user (such as a designer) to use, whereas the second one requires a bit of knowledge of PHP.

Also, notice in the second one, the require line is used to set up the variables `$METADATA` and `$PAGECONTENT`. A non-PHP programmer might be confused if they forgot to include that and there were empty spaces in the resulting HTML.

Another very important reason is that PHP files tend to be tied to specific URLs, such as `http://cms/page1.php`. If you have a number of different pages, and the designer wants to adjust the design, the designer needs to change all of those existing pages.

If the template is kept in an external page and interpreted through an engine, you get a number of advantages:

- Designers don't and can't write PHP in the template files. This makes the engine more robust, allowing the designers to do what they want without risking breakage.
- Programmers don't mess with template files. This means that there is no overlap between what the programmers and designers are doing, making the end product more stable than if they were constantly tweaking each others' code.
- Because the templates are external, you can swap designs by moving the theme directories around.

I think the most important aspect of this is the separation of concerns. The programmer (you) handles programming, the designer handles design, and the only time the work collides is when the design is being interpreted by the templating engine.

Writing your own templating engine is not hard. For a few years, I used my own, which was based on code similar to the previous examples.

One problem with home-grown templating engines is that they do not always have the robustness and speed of the more established engines.

Smarty speeds up its parsing by compiling the template into a PHP script, and then caching that script in a directory set aside for that purpose.

If you use an accelerator such as APC, ionCube, Zend, and so on, then that compiled script will then be cached in-memory by the accelerator, speeding it up even further.

There are other templating engines out there—Twig, FastTemplate, PHAML, and others. They all do basically the same thing, so which you use is perhaps a personal choice. For me, Smarty works and I've no reason to choose another. It's simple to use, and fast.

File layout of a theme

We've discussed how a templating engine works. Now let's look at a more concrete example.

1. Create a directory `/ww.skins` in the CMS webroot.
2. Within that directory, each theme has its own directory. We will create a very simple theme called "basic".
3. Create a directory `/ww.skins/basic`, and in that, create the following directories:

`/ww.skins/basic/h`	This will hold the HTML template files.
`/ww.skins/basic/c`	This will hold any CSS files.
`/ww.skins/basic/i`	This will hold images.

Usually, that will be enough for any theme. The only necessary directory there is `/h`. The others are simply to keep things neat.

If I wanted to add JavaScript specific to that theme, then I would add it to a `/ww.skins/basic/j` directory. You can see how it all works.

In this basic theme, we will have two templates. One with a menu across the top (horizontal), and one with a menu on the left-hand side (vertical). We will then assign these templates to different pages in the admin area.

In the admin area, the templates will be displayed in alphabetical order.

If there is one template that you prefer to be the default one used by new pages, then the template should be named `_default.html`. After sorting alphabetically, the underscore causes this file to be top of the list, versus other filenames which begin with letters.

`.html` is used as the extension for the template so that the designer can easily view the file in a browser to check that it looks okay.

Let's create a default template then, with a menu on the left-hand side. Create this file as `/ww.skins/basic/h/_default.html`:

```
<!doctype html>
<html>
  <head>
    {{$METADATA} }
    <link rel="stylesheet"
        href="/ww.skins/basic/c/style.css"/>
  </head>
```

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

The reason that {{ is used instead of {, is that it if the designer used the brace character ({) for anything in the HTML, or the admin used it in the page content, then it would very likely cause the templating engine to crash—it would become confused because it could not tell whether you meant to just display the character, or use it as part of a code.

By doubling the braces {{ ... }}, we reduce the chance of this happening immensely. Doubled braces very rarely (I've never seen it at all) come up in normal page text.

The reason we use braces at all, and not something more obviously programmatic such as "<!--{ ... }-->", is that it is readable. It is easier to read "insert {{$pagename}} here" than to read "insert <!--{$pagename}--> here".

I've introduced two variables and a function in the template:

{{$METADATA}}	This variable is an automatically generated string consisting of <head> child elements such as <title> and <script> tags to load jQuery, and so on.
{{$PAGECONTENT}}	This variable is the page body text.
{{MENU}}	This is a function which builds up a menu. It can have a number of options attached. You've seen "direction" in the template example code. We'll discuss this later in the chapter.

The template includes a hardcoded reference to the stylesheet.

We could insist that the stylesheet always be named /ww.skins/themename/c/styles.css, and that would allow us to include it automatically in the {{$METADATA}} variable, but we can't do this—different templates may need different styles that cause problems if they are within the one stylesheet.

Another reason is that if the stylesheet is in the template code, then the designer can work on the design without needing to load it through the CMS.

Setting up Smarty

Okay—we have a simple template. Let's display it on the front-end.

To do this, we first edit `/ww.incs/basics.php` to have it figure out where the theme is. Add this code to the end of the file:

```
// { theme variables
if(isset($DBVARS['theme_dir']))
    define('THEME_DIR',$DBVARS['theme_dir']);
else define('THEME_DIR',SCRIPTBASE.'ww.skins');
if(isset($DBVARS['theme']) && $DBVARS['theme'])
    define('THEME',$DBVARS['theme']);
else{
  $dir=new DirectoryIterator(THEME_DIR);
  $DBVARS['theme']='.default';
  foreach($dir as $file){
    if($file->isDot())continue;
    $DBVARS['theme']=$file->getFileName();
    break;
  }
  define('THEME',$DBVARS['theme']);
}
// }
```

In this, we set two constants:

`THEME_DIR`	This is the directory which holds the themes repository. Note that we leave the option open for it to be located somewhere other than /ww.skins if we want to move it.
`THEME`	The name of the selected theme. This is the name of the directory which holds the theme files.

The `$DBVARS` array, from `/.private/config.php`, was originally intended to only hold information on database access, but as I added to the CMS, I found this was the simplest place to put information which we need to load in every page of the website.

Instead of creating a second array, for non-database stuff, it made sense to have one single array of site-wide configuration options. Logically, it should be renamed to something like `$SITE_OPTIONS`, but it doesn't really matter. I only use it directly in one or two places. Everywhere else, it's the resulting defined constants that are used.

After setting up `THEME_DIR`, defaulting to `/ww.skins` if we don't explicitly set it to something else, we then set up `THEME`.

If no `$DBVARS['theme']` variable has been explicitly set, then `THEME` is set to the first directory found in `THEME_DIR`. In our example case, that will be the `/ww.skins/basic` directory.

Now we need to install Smarty.

To do this, go to `http://www.smarty.net/download.php` and download it. I am using version 2.6.26.

Unzip it in your `/ww.incs` directory, so there is then a `/ww.incs/Smarty-2.6.26` directory.

We do not need to use Smarty absolutely everywhere. For example, we don't use it in the admin area, as there is no real need to do templating there.

For this reason, we don't put the Smarty setup code in `/ww.incs/basics.php`.

Open up `/ww.incs/common.php`, and add this to the end of it:

```
require_once SCRIPTBASE
    . 'ww.incs/Smarty-2.6.26/libs/Smarty.class.php';
function smarty_setup($cdir){
  $smarty = new Smarty;
  if(!file_exists(SCRIPTBASE.'ww.cache/'.$cdir)){
    if(!mkdir(SCRIPTBASE.'ww.cache/'.$cdir)){
      die(SCRIPTBASE.'ww.cache/'.$cdir.' not created.<br />
          please make sure that '.USERBASE.'ww.cache is
          writable by the web-server');
    }
  }
  $smarty->compile_dir=SCRIPTBASE.'ww.cache/'.$cdir;
  $smarty->left_delimiter = '{{';
  $smarty->right_delimiter = '}}';
  $smarty->register_function('MENU', 'menu_show_fg');
  return $smarty;
}
```

As we'll see shortly, Smarty will not only be used in the theme's templates. It can be used in other places as well. To reduce repetition, we create a `smarty_setup()` function where common initializations are placed, and common functions are set up.

First, we make sure that the compile directory exists. If not, we create it (or `die()` trying).

We change the delimiters next to {{ and }}.

Also note the `MENU` function (you'll remember from the template code) is registered here. If Smarty encounters a `MENU` call in a template, it will call the `menu_show_fg()` function, which we'll define later in this chapter.

We do not define `$METADATA` or `$PAGECONTENT` here because they are explicitly tied to the page template.

Remove the last line (the `echo $PAGEDATA->body;` line) from `/index.php`.

We discussed how pages can have different "types". The `$PAGECONTENT` variable may need to be set up in different ways depending on the type, so we add a switch to the `index.php` to generate it:

```
// { set up pagecontent
switch($PAGEDATA->type){
  case '0': // { normal page
    $pagecontent=$PAGEDATA->body;
    break;
  // }
  // other cases will be handled here later
}
// }
```

That gets the page body and sets `$pagecontent` with it (we'll add it to Smarty shortly).

Next, we need to define the `$METADATA` variable. For that, we'll add the following code to the same file (`/index.php`):

```
// { set up metadata
  // { page title
  $title=($PAGEDATA->title!='')?
      $PAGEDATA->title:
      str_replace('www.','',$_SERVER['HTTP_HOST']).' > '
          .$PAGEDATA->name;
  $metadata='<title>'.htmlspecialchars($title).'</title>';
  // }
  // { show stylesheet and javascript links
  $metadata.='<script src="http://ajax.googleapis.com/ajax/
      libs/jquery/1.4.2/jquery.min.js"></script>'
    .'<script src="http://ajax.googleapis.com/ajax/libs/
      jqueryui/1.8.1/jquery-ui.min.js"></script>' ;
  // }
  // { meta tags
  $metadata.='<meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8" />';
  if($PAGEDATA->keywords)
```

```
        $metadata.='<meta http-equiv="keywords" content="'
            .htmlspecialchars($PAGEDATA->keywords).'" />';
    if($PAGEDATA->description)$metadata.='<meta
        http-equiv="description"
        content="'.htmlspecialchars($PAGEDATA->description).'"
        />';
    // }
  // }
```

If a page title was not provided, then the title is set up as the server's hostname plus the page name.

We include the jQuery and jQuery-UI libraries on every page.

The `Content-Type` metadata is included because even if we send it as a header, sometimes someone may save a web page to their hard drive. When a page is loaded from a hard drive without using a server, there is no `Content-Type` header sent so the file itself needs to contain the hint.

Finally, we add keywords and descriptions if they are needed.

Note that we added jQuery-UI, but did not choose one of the jQuery-UI themes. We'll talk about that later in this chapter, when building the page menu.

Next, we need to choose which template to show. Remember that we discussed how site designs may have multiple templates, and each page needs to select one or another.

We haven't yet added the admin part for choosing a template, so what we'll do is, similar to the `THEME` setup, we will simply look in the theme directory and choose the first template we find (in alphabetical order, so `_default.html` would naturally be first).

Edit `index.php` and add this code:

```
// { set up template
if(file_exists(THEME_DIR.'/'.THEME.'/h/'
    .$PAGEDATA->template.'.html')){
  $template=THEME_DIR.'/'.THEME.'/h/'
      .$PAGEDATA->template.'.html';
}
else if(file_exists(THEME_DIR.'/'.THEME.'/h/_default.html')){
  $template=THEME_DIR.'/'.THEME.'/h/_default.html';
}
else{
  $d=array();
```

```
    $dir=new DirectoryIterator(THEME_DIR.'/'.THEME.'/h/');
    foreach($dir as $f){
      if($f->isDot())continue;
      $n=$f->getFilename();
      if(preg_match('/^inc\./',$n))continue;
      if(preg_match('/\.html$/',$n))
          $d[]=preg_replace('/\.html$/','',$n);
    }
    asort($d);
    $template=THEME_DIR.'/'.THEME.'/h/'.$d[0].'.html';
  }
  if($template=='')die('no template created.
      please create a template first');
  // }
```

So, the order here is:

1. Use the database-defined template if it is defined and exists.
2. Use `_default.html` if it exists.
3. Use whatever is alphabetically first in the directory.
4. `die()`!

The reason we check for `_default.html` explicitly is that it saves time. We have set the convention so when creating a theme the designer should name the default template `_default.html`, so it is a waste of resources to search and sort when it can simply be set.

Note that we are ignoring any templates which begin with `"inc."`. Smarty can include files external to the template, so some people like to save the HTML for common headers and footers in external files, then include them in the template. If we simply add another convention that all included files must start with `"inc."` (for example, `inc.footer.html`), then using this code, we will only ever select a full template, and not accidentally use a partial file.

For full instructions on what Smarty can do, you should refer to the online documentation at `http://www.smarty.net/`.

Finally, we set up Smarty and tell it to render the template.

Add this to the end of the same file:

```
$smarty=smarty_setup('pages');
$smarty->template_dir=THEME_DIR.'/'.THEME.'/h/';
// { some straight replaces
$smarty->assign('PAGECONTENT',$pagecontent);
```

```
$smarty->assign('PAGEDATA',$PAGEDATA);
$smarty->assign('METADATA',$metadata);
// }
// { display the document
header('Content-type: text/html; Charset=utf-8');
$smarty->display($template);
// }
```

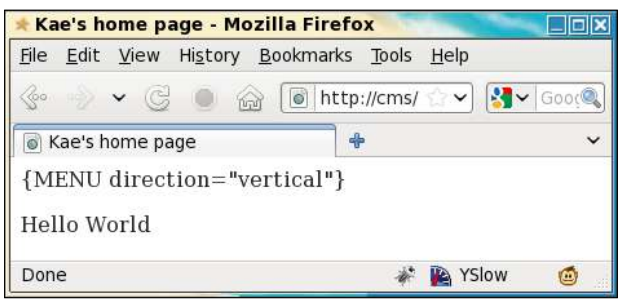
This section first sets up Smarty, telling it to use the `/ww.cache/pages` directory for caching compiled versions of the template.

Then the `$pagecontent` and `$metadata` variables are assigned to it.

We also assign the `$PAGEDATA` object to it, which lets us expose the page object to Smarty, in case the designer wants to use some aspect of it directly in the design. For example, the page name can be displayed with `{{$PAGEDATA->name|escape}}`, or the last edited date can be shown with `{{$PAGEDATA->edate|date_format}}`.

Before viewing this in a browser, edit the `/ww.skins/basics/_default.html` file, and change the double braces around the `MENU` call to single braces. We haven't yet defined that function, so we don't want Smarty to fail when it encounters it.

When viewed in a browser, we now have this screenshot:



It is very similar to the one from *Chapter 1*, *CMS Core Design*, except that we now have the page title set correctly.

Viewing the source, we see that the template has correctly been wrapped around the page content:

Okay—we can now see that the templating engine works for simple variable substitution. Now let's add in functions, and get the menu working.

Before going onto the next section, edit the template again and fix the braces so they're double again.

Front-end navigation menu

The easiest way to create a navigation menu is simply to list all the pages on the site. However, that does not give a contextual feel for where everything is in relation to everything else.

In the admin area, we created a hierarchical `<ul>` list. This is probably the easiest menu which gives a good feel. And, using jQuery, we can provide that `<ul>` list in all cases and transform it to whatever we want.

Let's start by creating the templating engine's `MENU` function with a `<ul>` tree, and we'll expand on that afterwards.

We've already registered `MENU` to run the function `show_menu_fg()`, so let's create that function.

We will add it to `/ww.incs/common.php`, where most page-specific functions go:

```
function menu_show_fg($opts){
  $c='';
  $options=array(
```

```
    'direction' => 0,  // 0: horizontal, 1: vertical
    'parent'    => 0,  // top-level
    'background'=> '', // sub-menu background colour
    'columns'   => 1,  // for wide drop-down sub-menus
    'opacity'   => 0   // opacity of the sub-menu
  );
  foreach($opts as $k=>$v){
    if(isset($options[$k]))$options[$k]=$v;
  }
  if(!is_numeric($options['parent'])){
    $r=Page::getInstanceByName($options['parent']);
    if($r)$options['parent']=$r->id;
  }
  if(is_numeric($options['direction'])){
    if($options['direction']=='0')
        $options['direction']='horizontal';
    else $options['direction']='vertical';
  }
  $menuid=$GLOBALS['fg_menus']++;
  $c.='<div class="menu-fg menu-fg-'.$options['direction']
      .'" id="menu-fg-'.$menuid.'">'
      .menu_build_fg($options['parent'],0,$options)
      .'</div>';
  return $c;
}
$fg_menus=0;
```

`menu_show_fg()` is called with an array of options as its only parameter. The first few lines of the function override any default values with values that were specified in the array (inspired by how jQuery plugins handle options).

Next, we set up some variables, such as getting details about the menu's parent page if there is one, and convert the direction to use words instead of numbers if a number was given.

Then, we generate an ID for the menu, to distinguish it from any others that might be on the page. This is stored in a global variable. In a more structured system, this might be stored in a static variable in a class (such as how `Page` instances are cached in the `/ww.php_classes/Page.php` file), but the emphasis here is on speed, and it's quicker to access a variable directly than to find a class and then read the variable.

Finally, we build a wrapper, fill it with the menu's `<ul>` tree, and return the wrapper.

The `<ul>` tree itself is built using a second function, `menu_build_fg()`, which we'll add in a moment.

Before doing that, we need to add a new method to the `Page` object. We will be showing links to pages, and need to provide a function for creating the right address. Edit `/ww.php_classes/Page.php` and add these methods to the `Page` class:

```
function getRelativeURL(){
  if(isset($this->relativeURL))return $this->relativeURL;
  $this->relativeURL='';
  if($this->parent){
    $p=Page::getInstance($this->parent);
    if($p)$this->relativeURL.=$p->getRelativeURL();
  }
  $this->relativeURL.='/'.$this->getURLSafeName();
  return $this->relativeURL;
}
function getURLSafeName(){
  if(isset($this->getURLSafeName))
      return $this->getURLSafeName;
  $r=$this->urlname;
  $r=preg_replace('/[^a-zA-Z0-9,-]/','-',$r);
  $this->getURLSafeName=$r;
  return $r;
}
```

The `getRelativeUrl()` method ensures that a page's link includes its parents and so on. For example, if a page's name is `page2` and it is contained under the parent page `page1`, then the returned string is `/page1/page2`, which can be used in `<a>` elements in the HTML.

The `getURLSafeName()` ensures that if the admin used any potentially harmful characters such as `!£$%^&*?` in the page name, then they are converted to `-` in the page name. When used in a MySQL query, the hyphen character `-` acts as a wildcard. So for example, if there is a page name "who are tom & jerry?", then the returned string is `who-are-tom---jerry-`. This method is commonly used in blog software where its desired that the page name is used in the URL.

Combined, these methods allow the admin to provide "SEO-friendly" page addresses without needing them to remember what characters are allowed or not. Of course, it means that there may be clashes if someone creates one page called "test?" and another called "test!", but those are rare and it is easy for the admin to spot the problem.

Back to the menu—let's add the `menu_build_fg()` function to `/ww.incs/common.php`. This will be a large function, so I'll explain it a bit at a time:

```
function menu_build_fg($parentid,$depth,$options){
  $PARENTDATA=Page::getInstance($parentid);
  // { menu order
  $order='ord,name';
  if(isset($PARENTDATA->vars->order_of_sub_pages)){
    switch($PARENTDATA->vars->order_of_sub_pages){
      case 1: // { alphabetical
        $order='name';
        if($PARENTDATA->vars->order_of_sub_pages_dir)
            $order.=' desc';
        break;
      // }
      case 2: // { associated_date
        $order='associated_date';
        if($PARENTDATA->vars->order_of_sub_pages_dir)
            $order.=' desc';
        $order.=',name';
        break;
      // }
      default: // { by admin order
        $order='ord';
        if($PARENTDATA->vars->order_of_sub_pages_dir)
            $order.=' desc';
        $order.=',name';
      // }
    }
  }
  // }
  $rs=dbAll("select id,name,type from pages where parent='"
      .$parentid."' and !(special&2) order by $order");
  if($rs===false || !count($rs))return '';
```

This first section gets the list of pages in this level of the menu from the database.

First, we get data about the parent page.

Next we figure out what sorting order the admin wanted that parent page's sub-pages to be displayed in, and we build up an SQL statement based on that.

Note the `and !(special&2)` part of the SQL statement. As explained in the previous chapter, we're using the `special` field as a bitfield. The `&` here is a Boolean `AND` function and returns true if the 2 bit is set (the 2 bit corresponds to "Does not appear in navigation"). So what this section means is "and not hidden".

If no pages are found, then an empty string is returned.

Now add this part of the function to the file:

```
$items=array();
foreach($rs as $r){
  $item='<li>';
  $page=Page::getInstance($r['id']);
  $item.='<a href="'.$page->getRelativeUrl().'">'
      .htmlspecialchars($page->name).'</a>';
  $item.=menu_build_fg($r['id'],$depth+1,$options);
  $item.='</li>';
  $items[]=$item;
}
$options['columns']=(int)$options['columns'];

// return top-level menu
if(!$depth)return '<ul>'.join('',$items).'</ul>';
```

What happens here is that we take the result set we got from the database in the previous section, and we build a list of links out of them using the `getRelativeURL()` method to generate safe URLs, and then display the admin-defined name using `htmlspecialchars()`.

Before each `<li>` is closed, we then recursively check `menu_build_fg()` with the current link as the new parent (the highlighted line). If there are no results, then the returned string will be blank. Otherwise it will be a sub-`<ul>` which will be inserted here.

If we are at the top level of the menu, then this generated list is immediately returned, wrapped in `<ul>...</ul>` tags.

The next section of code is triggered only if the call was to a sub-menu where `$depth` is 1 or more, for example from the call in the highlighted line in the last code section:

```
$s='';
if($options['background'])$s.='background:'
    .$options['background'].';';
if($options['opacity'])$s.='opacity:'
    .$options['opacity'].';';
if($s){
  $s=' style="'.$s.'"';
}

// return 1-column sub-menu
if($options['columns']<2)return '<ul'.$s.'>'
    .join('',$items).'</ul>';
```

This section checks to see if the `options` array had background or opacity rules for sub-menus, and applies them.

This is useful in the case that you are switching themes in the admin area, and the theme you switch to hasn't written CSS rules about sub-menus. It is very hard to think of every case that can occur, so designers sometimes don't cover all cases. As an example of this, imagine you have just created a new plugin for the CMS, and it looks good in a new theme designed specifically for it. The admin however, might prefer the general look of an older theme and selects it in the admin area (we'll get to that in this chapter). Unfortunately, that older theme does not have CSS rules to handle the new code.

In these cases, we need to provide workarounds so the code looks okay no matter the theme. In a later chapter, we'll look at how the menu options can be adjusted from the admin area, so that an admin can choose the sub-menu background color and opacity to fit any design they choose (in case the theme has not covered the case already).

The final line of the example returns the sub-menu wrapped in a `<ul>` element in the case that only one column is needed (the most common sub-menu type, and the default).

Now, let's add some code for multi-column sub-menus:

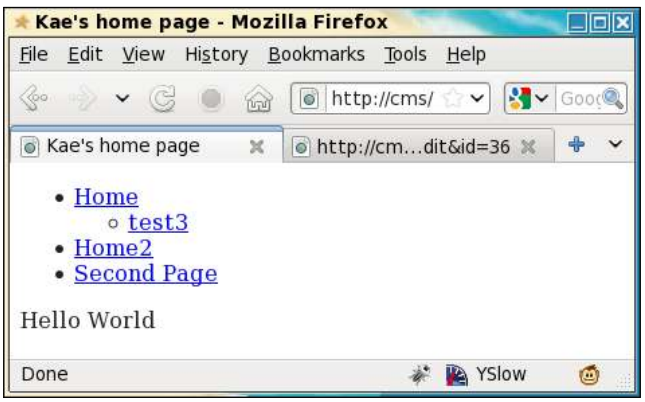
```
    // return multi-column submenu
    $items_count=count($items);
    $items_per_column=ceil($items_count/$options['columns']);
    $c='<table'.$s.'><tr><td><ul>';
    for($i=1;$i<$items_count+1;++$i){
      $c.=$items[$i-1];
      if($i!=$items_count && !($i%$items_per_column))
          $c.='</ul></td><td><ul>';
    }
    $c.='</ul></td></tr></table>';
    return $c;
}
```

In a number of places throughout the book, I've used HTML tables to display various layouts. While modern designers prefer to avoid the use of tables for layout, sometimes it is much easier to use a table for multi-columned layouts, then to try to find a working cross-browser CSS alternative. Sometimes the working alternative is too complex to be maintainable.

Another reason is that if we were to use a CSS alternative, we would be pushing CMS-specific CSS into the theme, which may conflict with the theme's own CSS. This should be avoided whenever possible.

In this case, we return the sub-menu broken into multiple columns. Most sites will not need this, but in sites that have a huge number of entries in a sub-menu and the sub-menu stretches longer than the height of the window, it's sometimes easier to use multiple columns to fit them all in the window than to get the administrator to break the sub-menu down into further sub-categories.

We can now see this code in action. Load up your home page in the browser, and it should look something like the next screenshot:



In my own database, I have two pages under `/Home`, but one of them is marked as hidden.

So, this shows how to create the navigation tree.

In the next chapter, we will improve on this menu using jQuery, and will then write a theme management system.

Summary

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

In the next chapter, we will improve the menu, write a theme management system, and add the ability to embed Smarty templating code within page content.