**Crash Course**

Templating basics: Assign content to Smarty, then display a template file.

|  |
| --- |
| **index.php**  include('Smarty.class.php');  // create object  $smarty = new Smarty;  // assign some content. This would typically come from  // a database or other source, but we'll use static  // values for the purpose of this example.  $smarty->assign('name', 'george smith');  $smarty->assign('address', '45th & Harris');  // display it  $smarty->display('index.tpl'); |

The template file then contains the presentational output interspersed with **{tags}** that Smarty replaces with dynamic content.

|  |  |  |
| --- | --- | --- |
| **index.tpl**  <html>  <head>  <title>Info</title>  </head>  <body>  <pre>  User Information:  Name: {$name}  Address: {$address}  </pre>  </body>  </html> |  | **output**  <html>  <head>  <title>Info</title>  </head>  <body>  <pre>  User Information:  Name: george smith  Address: 45th & Harris  </pre>  </body>  </html> |

As you can see, Smarty cleanly separates your presentation (HTML/CSS) from your application (PHP) code. However, it still contains some logic of its own. With respect to Smarty's design principles, *the logic in the templates should be for presentation only*. For instance, looping over table row colors or including one template from another would be considered presentation logic. This way the application code only needs to supply the content, and the template can take care of the presentation. The idea is to keep the template designer role (front-end developer) and application programming role (backend-developer) separated. You should be able to completely redesign the presentation (of the content) without altering the application code. You should also be able to restructure the application code and use the same templates, just keep the content assigned to the same variables. This design goes a long ways toward maintenance and stability.

Sometimes alteration of the assigned content is required for presentation. This is accomplished with *variable modifiers.* These are used to alter the *output* of assigned variables from within the template. In our example, we would like to display George's name capitalized, and we would like to properly HTML escape the amphersand (&) symbol in the address. We also show how to display the current date with custom formatting.

|  |  |  |
| --- | --- | --- |
| **index.tpl**  <html>  <head>  <title>Info</title>  </head>  <body>  <pre>  User Information:  Name: {$name|capitalize}  Addr: {$address|escape}  Date: {$smarty.now|date\_format:"%b %e, %Y"}  </pre>  </body>  </html> |  | **output**  <html>  <head>  <title>Info</title>  </head>  <body>  <pre>  User Information:  Name: George Smith  Addr: 45th &amp; Harris  Date: Jul 16, 2010  </pre>  </body>  </html> |

Using modifiers, we have just handed formatting control of the assigned content over to the template. The idea is to assign the unmodified content to the template, keeping presentation logic out of your application code.

You can chain any number of modifiers together on one variable, making this feature quite flexible. There are many more [modifiers](http://www.smarty.net/docs/en/language.modifiers.php) that come with Smarty, or you can make your own with its easy to use [plugin](http://www.smarty.net/docs/en/plugins) architecture. Drop your new modifier into the plugin directory, then mention it in the template!

You can also use or create *template functions*. Template functions carry out tasks in the template. For example, you can include other templates with the *{include}* function, you can generate a set of select options with the *{html\_options}* function, or you can cycle through a set of values with the *{cycle}* function. Let's say you have many templates with the same header and footer information. You can manage these as separate templates and include them.

|  |  |  |
| --- | --- | --- |
| **header.tpl**  <html>  <head>  <title>{$title|default:"no title"}</title>  </head>  <body> |  | **footer.tpl**  </body>  </html> |
| **index.tpl**  {include file="header.tpl" title="Info"}  User Information:<p>  Name: {$name|capitalize}<br>  Address: {$address|escape}<br>  {include file="footer.tpl"} |  | **output**  <html>  <head>  <title>Info</title>  </head>  <body>  User Information:<p>  Name: George Smith<br>  Address: 45th &amp; Harris<br>  </body>  </html> |

One feature of the {include} function is locally scoped variables. Notice that *$title* is a template variable not assigned directly to the template, but assigned by passing it as an attribute to the *{include}* function. This way the *$title* variable is only available within the scope of the header template, and can be dynamically changed any time the header.tpl file is included. Also, notice we used the *default* modifier on *$title* so in the case $title is empty, the text "*no title*" will show up instead of displaying nothing.

An alternate approach to managing template hierarchy is *Template Inheritance*. A template can inherit another, and change specific blocks of content within that template. For example:

**parent.tpl**

<html>

<head>

<title>{block name=title}Default Title{/block}</title>

</head>

<body>

{block name=body}Default Body{/block}

</body>

</html>

**child.tpl**

{extends file="parent.tpl"}

{block name=title}My Title{/block}

{block name=body}My Body{/block}

**output**

<html>

<head>

<title>My Title</title>

</head>

<body>

My Body

</body>

</html>

There are some nice functions that automate tasks such as html dropdown boxes. One is *html\_options*. You assign the arrays of data to the template, then this function does all the work to generate the HTML output for it.

|  |
| --- |
| **index.php**  include('Smarty.class.php');  // create object  $smarty = new Smarty;  // assign options arrays  $smarty->assign('id', array(1,2,3,4,5));  $smarty->assign('names', array('bob','jim','joe','jerry','fred'));  // display it  $smarty->display('index.tpl'); |

|  |
| --- |
| **index.tpl**  <select name=user>  {html\_options values=$id output=$names selected="5"}  </select> |

|  |
| --- |
| **output**  <select name=user>  <option label="bob" value="1">bob</option>  <option label="jim" value="2">jim</option>  <option label="joe" value="3">joe</option>  <option label="jerry" value="4">jerry</option>  <option label="fred" value="5" selected="selected">fred</option>  </select> |

You can quickly loop over arrays of data with the *foreach* function. Here's an example of that, and we also threw in alternating row colors via the *{cycle}* function, and we use the *{strip}* function to strip out extra white space.

|  |
| --- |
| **index.php**  include('Smarty.class.php');  // create object  $smarty = new Smarty;  // assign an array of data  $smarty->assign('names', array('bob','jim','joe','jerry','fred'));  // assign an associative array of data  $smarty->assign('users', array(  array('name' => 'bob', 'phone' => '555-3425'),  array('name' => 'jim', 'phone' => '555-4364'),  array('name' => 'joe', 'phone' => '555-3422'),  array('name' => 'jerry', 'phone' => '555-4973'),  array('name' => 'fred', 'phone' => '555-3235')  ));  // display it  $smarty->display('index.tpl'); |

|  |
| --- |
| **index.tpl**  <table>  {foreach $names as $name}  {strip}  <tr bgcolor="{cycle values="#eeeeee,#dddddd"}">  <td>{$name}</td>  </tr>  {/strip}  {/foreach}  </table>  <table>  {foreach $users as $user}  {strip}  <tr bgcolor="{cycle values="#aaaaaa,#bbbbbb"}">  <td>{$user.name}</td>  <td>{$user.phone}</td>  </tr>  {/strip}  {/foreach}  </table> |

|  |
| --- |
| **output**  <table>  <tr bgcolor="#eeeeee"><td>bob</td></tr>  <tr bgcolor="#dddddd"><td>jim</td></tr>  <tr bgcolor="#eeeeee"><td>joe</td></tr>  <tr bgcolor="#dddddd"><td>jerry</td></tr>  <tr bgcolor="#eeeeee"><td>fred</td></tr>  </table>  <table>  <tr bgcolor="#aaaaaa"><td>bob</td><td>555-3425</td></tr>  <tr bgcolor="#bbbbbb"><td>jim</td><td>555-4364</td></tr>  <tr bgcolor="#aaaaaa"><td>joe</td><td>555-3422</td></tr>  <tr bgcolor="#bbbbbb"><td>jerry</td><td>555-4973</td></tr>  <tr bgcolor="#aaaaaa"><td>fred</td><td>555-3235</td></tr>  </table> |

Smarty also has built-in caching capabilities to help speed up the page rendering. A copy of the template output is stored in a text file, then that is displayed upon subsequent calls to the request instead of dynamically rendering the page each time. This can speedup page rendering substantially, especially if there is a lot of processing involved to create the page such as database calls and variable assignments. You can also leave parts of a page dynamic just by marking them as nocache.

|  |
| --- |
| **index.php**  include('Smarty.class.php');  // create object  $smarty = new Smarty;  $smarty->setCaching(true);  // see if the page is already cached  if(!$smarty->isCached('index.tpl')) {  // not cached, so you might do some database queries here,  // then assign the returned content. We just use static  // values for this example.  $smarty->assign('name', 'george smith');  $smarty->assign('address', '45th & Harris');  }  // display it  $smarty->display('index.tpl'); |

You can also have multiple cached versions of a single page by passing a unique cache id to the display() function.