Template Class

Introduction

In ExpressionEngine, a template is effectively taking the place of a page on a site and when rendered displays content from modules and plugins. Whenever a page is requested within an ExpressionEngine site, the system calls the Template class to discover which template is being requested and then does all of the parsing and processing required for rendering the template. This processing includes the embedding of other templates, tag and template caching, PHP processing, and it performs the calling of modules and plugins when their tags are found in the template.

A typical module or plugin tag in ExpressionEngine looks like this:

```
{exp:channel:channel_name}
```

The first part of the tag, **exp:**, tells ExpressionEngine that this is a tag. The second part, **channel**, is the module or the plugin that the tag belongs to, in this case the "channel" module. The third part is the specific method from within the module or plugin that you're calling; in this case the function to display the "channel_name".

Calling the Template Class

When the Template class calls a module, the Template class makes available to the module class the parameters, variables, and content of the tag currently being processed. To make these values available, simply access the ***TMPL*** object of the ExpressionEngine super object.

```
function module_example()
{
        // Make a local reference to the ExpressionEngine super object
        $this->EE =& get_instance();

        $name = $this->EE->TMPL->fetch_param('name', '');
}
```

Parameters

Before calling the function that is being requested by the ExpressionEngine tag, the Template class will parse out any parameters for that tag and insert them into a Template class variable, thus making them easily accessible by the requested function by using the **fetch_param()** method.

string **$this->EE->TYPE->fetch_param** (string name, [string default])

fetch_param() takes an optional second argument, which is the value that will be returned if the parameter was not set in the opening tag. It defaults to FALSE if the second argument is not supplied.

Note: values of 'y', 'on' and 'yes' will all return 'yes', while 'n', 'off' and 'no' all return 'no'.

```
// Default value is 'random'
```

```
$order = $this->EE->TMPL->fetch_param('order', 'random');

if ( ! $channel = $this->EE->TMPL->fetch_param('channel'))
{
    $this->EE->language->fetch_language_file('module');
    $this->EE->output->fatal_error($this->EE->language->line('module_invalid_channel'));
    exit;
}
else
{
    $str = $this->EE->functions->sql_andor_string($channel, 'channel_name');

    if (strncmp($str, 'AND', 3) == 0)
    {
        $str = substr($str, 3);
    }

    $sql .= "SELECT channel_id FROM exp_channels WHERE ".$str;
    $query = $this->EE->db->query($sql);
}
// If channel is not specified, then an error is output.
// Otherwise, perform query.
```

Information Within Tag Pairs

ExpressionEngine tags are used primarily to output some form of content so it can be displayed within a template. The formatting for this content is determined by the HTML and variable data contained between the opening and closing tags for the tag being called. We normally call this formatting information between the opening and closing tags the 'tag data', and this data can be requested by using the $this->EE->TMPL->tagdata variable.

Note: *Except in rare cases, a module will have both an opening and closing tag. There are exceptions to this rule such as when you might wish to have a tag that simply performs an automated action. An example of this is the {exp:moblog:check} tag.*

Module code in template. The tag data is everything from the end of the opening tag to the beginning of the closing tag, basically the HTML and tag variables:

```
{exp:magic:spell}

<h2>{title}</h2>

<p>{summary}</p>

{/exp:magic:spell}
```

A module calling and using the tag data.

```
$query = $this->EE->db->query($sql);
$variables = array();

foreach($query->result as $row)
{
    $variables[] = array(
                        'foo' => $row['foo'],
                        'bar' => $row['bar']
                        );
}

return $this->EE->TMPL->parse_variables($tagdata, $variables);
```

Variable Types

ExpressionEngine variables are simply a word or underscored phrase with curly brackets on either side. The names are usually quite simple and contextually understandable for the tag, thus making it easier for users to remember them and understand their usage. There are three kinds of variables in ExpressionEngine, single, pair, and conditional variables.

```
// Single Variable
{summary}

// Pair Variable
{category}

{/category}

// Conditional Variable
{if body != ""}

{/if}
```

Parsing Variables

Overview

The Template class makes parsing your module or plugin's variables a snap. Using the **parse_variables()** method, you supply the tag data, and an array containing all of your variables, organized as "rows". Your single, pair, and conditional variables will automatically be parsed for you, and your module or plugin will also automatically have {count} and {switch} variables. Additionally, date variables will be parsed, and you can optionally have typography performed automatically for you as well.

Master Variables Array

First let's look at a typical variables array:

```
Array
(
    [0] => Array
        (
            [powers] => Array
                (
                    [0] => Array
                        (
                            [power] => Super Strength
                            [scale] => 8
                        )

                    [1] => Array
                        (
                            [power] => Invisibility
                            [scale] => 4
                        )

                )

            [name] => Chameleon
            [dob] => 136771200
            [type] => Hero
            [affiliation] => Litigation Coalition
            [bio] => Array
                (
                    [0] => Hailing from the planet Lizzon, Chameleon came to earth in 2003.
                    [1] => Array
                        (
                            [text_format] => xhtml
                            [html_format] => all
                        )

                )

        )

    [1] => Array
        (
            [powers] => Array
                (
                    [0] => Array
                        (
                            [power] => Poisonous Breath
                            [scale] => 5
                        )

                    [1] => Array
                        (
                            [power] => Wealth
                            [scale] => 7
                        )

                )

            [name] => Stinkor
            [dob] => -58924800
            [type] => Villain
            [affiliation] => N.E.S.T.
            [bio] => Array
                (
                    [0] => As a child, Stinkor was teased for his bad breath. When he realized that it was more than bad...noxious even, he turned to
                    [1] => Array
                        (
                            [text_format] => xhtml
                            [html_format] => all
                        )

                )
        )

)
```

Looking at this example, we see two "rows" of results. Each "row" contains a pair variable, 'powers', which itself has multiple rows with some single variables, 'power' and 'scale'. Next we have the single variables 'name', 'dob', 'type', 'affiliation', and 'bio'. We can tell by looking that 'dob' is a date field, in this case date of birth. The 'bio' field, though a single variable is also an array, containing the contents and typography formatting instructions, but more on that later. Let's look at a typical way that this array would have been created in an add-on's code.

```
$variables = array();

foreach ($query->result as $row)
{
        $powers = array()

        foreach ($unserialize($row['powers']) as $power)
        {
                $powers[] = array('power' => $power['name'], 'scale' => $power['scale']);
        }

        $variable_row = array(
                                'powers'        => $powers,
                                'name'          => $row['name'],
                                'dob'           => $row['dob'],
                                'type'          => $row['type'],
                                'affiliation'   => $row['affiliation']
                                );

        $type_prefs = array('text_format' => 'xhtml', 'html_format' => 'all');

        $variable_row['bio'] = array($row['bio'], $type_prefs);

        $variables[] = $variable_row;
}
```

In the example above, first the pair variable $powers array is created. Each "row" of the pair variable is an array of single variables, or even more pair variables. Then an array is used to hold the data for this result's row. The simple single variables are added in a simple array() declaration, but bio, which needed some typography preferences, is added later as an additional key. Whether you use an array() declaration, or keys for assignment is entirely up to you, and will often depend on the needs of your code. At the end of the loop, we add the entire "row" of data to our master $variables array. That row is now stored for parsing.

Note that the order in which the variables are given in the array is the same order they will be parsed in. Because of this precedence, it is often best to place your pair variable arrays first.

Parsing the Variables

Now that our master array is fully loaded, we simply send it along with the tagdata to the **parse_variables()** method of the Template class, which returns the parsed output.

```
$output = $this->EE->TMPL->parse_variables($this->EE->TMPL->tagdata, $variables);
```

Assuming that our tagdata is as follows:

```
<h1>{name}</h1>
<ul>
        <li>Date of Birth: {dob format="%d %M, %Y"}</li>
        <li>{type}</li>
        <li>Affiliation: {affiliation}</li>
</ul>

<ul>
{powers}
        <li{if scale > 5} class="great"{/if}>{power} ({scale})</li>
{/powers}
</ul>

{bio}
```

Our returned output would be:

```
<h1>Chameleon</h1>
<ul>
        <li>Date of Birth: 02 May, 1974</li>
        <li>Hero</li>
        <li>Affiliation: Litigation Coalition</li>
</ul>

<ul>
        <li class="great">Super Strength (8)</li>
        <li>Invisibility (4)</li>
</ul>
```

```
<p>Hailing from the planet Lizzon, Chameleon came to earth in 2003.
</p>

<h1>Stinkor</h1>
<ul>
        <li>Date of Birth: 18 Feb, 1968</li>
        <li>Villain</li>
        <li>Affiliation: N.E.S.T.</li>
</ul>

<ul>
        <li>Poisonous Breath (5)</li>
        <li class="great">Wealth (7)</li>
</ul>

<p>As a child, Stinkor was teased for his bad breath.  When he realized that it was more than bad...noxious even, he turned to a life of crime, robbin
</p>
```

The following subsections break down the procedures in detail.

Single Variables

```
<h1>{name}</h1>
<ul>
        <li>Date of Birth: {dob format="%d %M, %Y"}</li>
        <li>{type}</li>
        <li>Affiliation: {affiliation}</li>
</ul>
```

Single variables are defined in the array as simple key => value pairs.

```
$vars = array(
                'name' => 'Stinkor',
                'type' => 'Villain',
                'dob' => -58924800,
                'affiliation' => 'N.E.S.T.'
                );
```

Additionally, you can have Typography automatically performed on single variables, by sending the variable in the form of an array with two keys - the first being the content, and the second being an array including any of the four available standard Typography preferences that you wish to override. Sending an empty array will result in Typography being parsed with the class defaults.

```
$type_prefs = array(
                'text_format'   => 'markdown',
                'html_format'   => 'all',
                'auto_links'    => 'y',
                'allow_img_url' => 'y'
                );

$vars['bio'] = array('This is the variable contents', $type_prefs);
```

Pair Variables

```
<ul>
{powers}
        <li>{power} ({scale})</li>
{/powers}
</ul>
```

Pair variables are defined identically to single variables, but contained in a multidimensional array of "rows" with the pair variable's name as the key.

```
$vars['powers'] = array(
                        array('power' => 'Poisonous Breath', 'scale' => 5),
                        array('power' => 'Wealth', 'scale' => 7),
                        array('power' => 'Flying', 'scale' => 6)
                        );
```

Automatic Variables

If you are using the parse_variables() method to handle variable parsing in your add-on, then your tag will automatically inherit the ability to use the following variables:

```
{count}
```

The "count" of the output; the iteration of the tag pair loop.

```
{total_results}
```

The total number of results, or "rows", that your tag will be outputting.

```
{switch="one|two|three"}
```

This variable permits you to rotate through any number of values as the results are displayed. The first result will use "option_one", the second will use "option_two", the third "option_three", the fourth "option_one", and so on.

Date Variables

When the Template Parser encounters a variable with a date formatting parameter, it will automatically format the variable for you, so it is important to send date variables as UTC/GMT Unix timestamps. Localization will automatically occur according to the site and logged in user's preferences.

```
$var['dob'] = -58924800;  // Nov 14, 1971 (UTC/GMT)
```

Conditionals

Your variables will automatically be made available to conditionals. No special processing is necessary in your add-on to handle conditionals for variables you send to the parser.

Parsing Full Results

Once you have assembled your master array of result "rows", with each row containing the single and pair variables that your tag uses, simply call the parse_variables() method, providing the tag data, and the master array.

```
$str = $this->EE->TMPL->parse_variables($tagdata, $variables);
```

Parsing a Single Result Row

You may also parse the result rows yourself, which could be useful if for some reason you need to modify the tagdata for each row based on certain criteria. You can still benefit from the simplified variable parsing by using parse_variables_row(), though you will no longer automatically have {count}, {total_results}, or {switch=} variables. To include these variables when parsing your own result rows, you will need to add them yourself.

```
$count = 0;
$output = '';

foreach($query->result as $row)
{
        $row['count'] = ++$count;
        $row['total_results] = $query->num_rows;

        $output .= $this->EE->TMPL->parse_variables_row($tagdata, $row);
}
```

Single and Pair Variables (Legacy)

Before calling the module for the ExpressionEngine tag, the Template class parses out all of the variables contain in the tag's data and puts them into arrays which are Template class variables. This allows the module to have a list of all the single, pair, and conditional variables that it needs to replace with content.

Single variables output a single piece of content, and in the module's code these variables are usually handled by doing a simple find and replace, where the outputted content is replacing the variable. The Template class array for single variables is $this->EE->TMPL->var_single, where the keys are the variable's name and the values are the full variable contents including any formatting parameters. For dates using format="%Y %m %d", only the formatting string is assigned to the array value. The Template class also provides a function, $this->EE->TMPL->swap_var_single, for performing the find and replace, making sure that the variable is replaced correctly in the template.

```
foreach ($this->EE->TMPL->var_single as $key => $val)
{
    if ($key == "spell_name")
    {
        $tagdata = $this->EE->TMPL->swap_var_single($val, $row['spell_name'], $tagdata);
    }

    if (strncmp($key, "spell_date", 10) == 0)
    {
        $date = $this->EE->localize->decode_date($val, $row['spell_date']);

        $tagdata = $this->EE->TMPL->swap_var_single($key, $date, $tagdata);
    }
}
```

Pair variables are a bit more complicated since they are often used for performing a loop within the tag data when there are multiple pieces of content of a similar type. A good example of this is the channel module where an entry might have multiple categories.

```
{exp:channel:entries}

<ul>
```

```
{categories}
<li>{category_name}</li>
{/categories}
</ul>

{exp:channel:entries}
```

The Template class variable containing the variable pairs in the tag data is $this->EE->TMPL->var_pair, which is an array where the keys are the contents of the pair variable's opening tag and the values are an array containing any parameters for the pair variable. Since the $this->EE->TMPL->var_pair variable does not contain the content of the variable pair, you will have to search the template for it yourself using a preg_match() (or possibly a preg_match_all(), if you believe there could be multiple instances of this variable pair).

```
foreach ($this->EE->TMPL->var_pair as $key => $val)
{
        if (strncmp($key, 'items', 5) == 0)
    {
        $temp = preg_match("/".LD.$key.RD."(.*?)".LD.'\'.SLASH.'items'.RD."/s", $this->EE->TMPL->tagdata, $matches)

        // Set the display preference
        $nest = (is_array($val) && isset($val['nest'])) ? $val['nest'] : 'no';

        if ($nest == 'yes')
        {
                $temp = $this->nested_items($this->items, $temp);
        }
        else
        {
                $temp = $this->linear_items($this->items, $temp);
        }
    }
}
```

Conditional Variables

Conditional variables allow scripting to be added to your module's tag data in order to show data if certain defined criteria are met. The structure should be a variable being checked against another variable or value via an operator:

```
// Structure
{if variable comparison-operator value}

Data between the tags that gets shown if the condition is met.

{/if}

// Example
{if spell_level > 3}

Advance Magicians Only

{/if}
```

There is a great deal more information about possible conditionals in the Conditional Global Variables, so we suggest you give it a quick look over.

If you are scripting conditional variables in your module, then they should be done first when processing tag data before any other variables are parsed. Instead of writing your own conditional parsing routine, ExpressionEngine allows you to simply give your data to a function that then takes care of all the work. The data needs to be in the form of an array where the key is the name of the variable and the value is the data for that variable.

If you have *short conditionals* that can be evaluted without a comparison operator (ex: {if allow_comments}), then instead of sending data you will send a string of either 'TRUE' or 'FALSE' depending on whether that conditional should be evaluated as true or false. The example belows gives you an idea of how this should work:

```
$cond                           = $row;         // $row contains query fields and values, ex:  'title' => "First Entry"

$cond['logged_in']              = ($this->EE->session->userdata('member_id') == 0) ? 'FALSE' : 'TRUE';
$cond['logged_out']             = ($this->EE->session->userdata('member_id') != 0) ? 'FALSE' : 'TRUE';
$cond['allow_comments']         = (isset($row['allow_comments']) AND $row['allow_comments'] == 'n') ? 'FALSE' : 'TRUE';

$tagdata = $this->EE->functions->prep_conditionals($tagdata, $cond);
```

Once you send your tag data and your array of conditional variables, the $this->EE->functions->prep_conditionals() function processes the conditionals so that they can be evaluted by the Template parser later.

Top of Page

User Contributed Notes

Posted by: Philip Zaengle on 30 October 2010 8:48am [Permalink]

When writing addons no_results is not handled with the normal $cond[] array, you actualy need to use the following which will trigger no results

```
if($results == 0){ // your no_results logic
  return $this->EE->TMPL->no_results();
}
```

Posted by: Jamie Rumbelow on 2 May 2010 7:38am [Permalink]

The EE2 template parser doesn't currently support path-based variables. As the “simple variable” replacement technique uses str_replace(), not preg_replace(), you need to manually parse any path variables:

```
$path = $this->EE->functions->create_url('something/else');
$tagdata = preg_replace("/".LD."name_path=(.*?)".RD."/", $path, $tagdata);
```

Taking a look at **expressionengine/libraries/Template.php** at line 3970, there's some sort of plan for path-based variables in there. We'll have to wait!

You must have an ExpressionEngine license and have attained a forum rank of "Lab Assistant" (100 posts) to contribute notes to the User Guide

User Guide Contributions Feed