FEATURES

Using, Reusing and Extending Smarty

By Tomica Jovanovic

Smarty represents an excellent solution for creating websites that are easy to manage and maintain, all the while providing very good performance and caching features. This article illustrates how you can get Smarty to give you more by extending its functionality.

The Question

When PHP was born, it was a very simple language, whose main purpose was to generate HTML code for the presentation of data. That data came from C functions that formed the application logic. One of the goals of PHP from the outset was to separate front-end "presentation" from the back-end "logic" of web applications.

However, by version 3 PHP had already become a pretty complex and complete language. With numerous string manipulating functions, arrays, and access to flat-file and SQL databases, PHP soon grew to be powerful enough to create complex, database driven sites. All you needed to create a small to medium sized dynamic web site was PHP, and a database server (usually MySQL).

Once people became daring enough to start the development of larger sites with PHP, however, it became evident that the old concept of mixing PHP and HTML code was not A Good Thing™. A quick solution was provided in the form of a PHP port of the popular FastTemplates engine, which was originally done in Perl.

Shortly thereafter, even though FastTemplates was probably the most popular template engine, a lot of others were created, since FastTemplates wasn't everyone's cup of tea (even I created my own system for this very reason). Most of the systems featured variable substitution, dynamic (repeating) blocks, inclusion of other templates, and sometimes even optional 'if' blocks. This was fine, except that almost all of them did this using regular expressions, which complicated code, and added additional overhead to the whole application.

They were hard to extend (and to adjust to taste), and didn't provide the complete separation of presentation and logic (for example, to make rows of a table alternate colors, PHP code had to be adjusted).

The Answer

The SmartTemplate system was the predecessor of today's 'Smarty', though it is not a direct ancestor. Development of SmartTemplate started in 1999, and it was very similar to all other template engines available at that time. Even though it supported some

REQUIREMENTS

PHP Version: **4.0.6 and above (4.1 recommended)**
Additional Software: **Smarty Template Engine (http://smarty.php.net)**

advanced features like configuration files and nested blocks, it suffered from the same issues as other template engines of that time. However, even though it had problems, it helped the creators gain an intimate knowledge of all aspects of the creation and use of template systems. The idea to employ a completely new concept was born from this experience, and this concept was welded together with the other ideas of the development team to form what later became Smarty.

This new revolutionary concept is referred to as "compiling" templates. It combines the speed of execution of pure PHP code with the ease and simplicity of template syntax. Using this process, template files are first converted to regular PHP code before they are executed. This sounds costly in terms of performance until you consider that this need only be done when the template file is changed! Once a template file is compiled, they are stored for later use (and re-use). The end result being a reduced overall server load.

The old concept of mixing PHP and HTML code is not A Good Thing™.

Why Not Smarty?

Some opponents of Smarty will say that it is too complex, that it has complicated syntax, and an overabundance of functions and features. Some have said that Smarty is a framework of its own, or even a language of its own, and that it isn't worthwhile to learn YAL (yet another language).

While I cannot confirm or deny those allegations, what I can do is offer a view from a different perspective. I already mentioned that Smarty (as PHP itself) was created to be used in the "presentation layer" of web applications. For comparison's sake let's look briefly at the opposite end of the multi-tier application architecture, at data storage. It wasn't long before creators and users of RDBMS realized that standard SQL wasn't enough to satisfy the needs of complex applications. That's why they created views, triggers, and even completely new languages (PL/SQL, T-SQL) for stored procedures and other tools necessary to maintain and insure data integrity. The task of those new languages and technologies was to separate the job of data storage from the application logic itself, just as the task of Smarty is to separate the job of presetting data from the main logic of the web application. This is not a trivial task to do thoroughly and accurately, which is why a somewhat complex system like Smarty is needed.

Extending Smarty

There are lots of Smarty tutorials on the net, because Smarty itself is gaining popularity. The most prevalent shortcoming in the majority of the tutorials is that they just barely scratch the surface in terms of showing off all of the functionality and the ways in which Smarty can be leveraged to get a job done. Even those which explain the idea of template compiling often say nothing about the second biggest advantage of Smarty, which is its extensibility.

It is this unique extensibility that really makes Smarty an invaluable development tool and solution framework. With Smarty's simple-yet-powerful 'plugin system', developers can truly have the flexibility they need to satisfy any requirement that may arise during a given project's life cycle.

The primary goal of this article is to explain some of Smarty's more advanced functionality, and to show you some real life examples of how you can elegantly use Smarty to ease your life (ok, your work then). If you don't have any experience with Smarty (or at least with a template system), I suggest that you read some introductory material on Smarty before you continue with this article.

Smarty basically processes templates. It uses template "tags" to distinguish dynamic content (data provided from your PHP application) from the rest of the template HTML. These tags are replaced with the result of some Smarty functions. Except for a couple of basic, or "core" Smarty functions, every other function is really included in the form of a plugin. This allows you to change the behaviour of current plugins, and to even add entirely new ones if you need them. This is what I plan to show you next, but first, let's set up some basic code.

As previously noted, for the sake of brevity, this article will assume a basic knowledge of Smarty or some other template system. That said, it quite naturally follows that I won't explain how to use Smarty, or its every function. In the examples to follow, I will explain only the interesting parts of code that relate directly to this article. Also, except in Listing 1, where a Smarty class is used, I will not use OOP programming techniques in the examples. I feel that it would only confuse someone not familiar with OOP concepts. On the other hand, those of you who understand OOP shouldn't have much difficulty converting all of these ideas into classes and objects. So let's dig in!

In Listing 1, you can see the basic code used in this

Listing 1: Basic Smarty include script - smarty.inc.php

```
1  <?php
2
3     include_once('proteus/smarty/Smarty.class.php');
4
5  function smarty($file, $vars) {
6       global $smarty_debugging, $error_log, $debug_log;
7       $smarty=new Smarty;
8       $smarty->template_dir='templates';
9       $smarty->compile_dir='templates_c';
10      $smarty->plugins_dir=array('my_plugins', 'plugins');
11      $smarty->debug_tpl='smarty_debug.htm';
12      $smarty->debugging=$smarty_debugging;
13      $smarty->load_filter('output', 'literal_cleanup');
14      $smarty->assign($vars);
15      $smarty->assign('error_log', $error_log);
16      $smarty->assign('debug_log', $debug_log);
17      $smarty->display($file);
18 }
19
20 ?>
```

article. This code will be modified later to demonstrate Smarty's extensibility. However, we'll view it in it's 'pristine' state here as a starting point. Keep this listing handy, as we'll be referring back to it later in the article

Basically, this function processes a template passed in using the variable **$file**, and passes data from an associative array **$vars** to the template engine. The first thing you should notice in Listing 1 is on line 10, where an array is assigned to the plugins_dir Smarty property. The second item in the array is the name of the plugin directory that comes with Smarty by default. The first item is the name of the plugin directory where I keep all of the plugins I have made or modified. It is smart to separate them like this, because when new versions of Smarty are installed, all I need to do is overwrite the standard files and directories, while my files remain safely in their own directory. Ordering of plugin directories in the Smarty **$plugin_dir** array is also important. For example, if you want to modify some of the standard Smarty plugins, you can just copy them to your plugin directory, and change them at will. Later, your version of the plugin will be used, because it is in the first plugins directory (in Smarty's plugin path).

Listing 2: Basic **substr()** Modifier - modifier.substr.php

```
<?php

function smarty_modifier_substr($string, $start,
                                 $length = 'dummy')
{
    if ($length==='dummy')
        return substr($string, $start);
    return substr($string, $start, $length);
}

?>
```

Smarty supports several types of plugins. Some plugins, such as template functions, modifiers and inserts, Smarty can load on its own if you use them in the template code. Others, such as Pre, Post and Output filters have to be explicitly loaded (with the **load_filter()** method), because they aren't mentioned in the template code. I will explain modifiers and filters here, and give a couple of very useful examples that show how easy it is to extend Smarty.

Modifiers are "small functions" that modify a variable that precedes them. To apply some modifier to a value, you append | (pipe) to a value, and then a name of a modifier. For example:

```
<b>{$article_title|capitalize}</b>
```

Modifiers can be combined (chained) together, and can accept parameters that are separated with a : (colon).

Writing your own modifier is not a difficult task. I often needed a classic PHP function - **substr()** in my templates, so I have built my own. Listing 2 shows code for a **substr()** modifier.

The official name of this modifier is "substr". Names of modifier functions need to start with "smarty_modifier_", followed by the actual modifier name. Modifier code must be placed in a file that resides in the plugin directory, and its name should start with "modifier", followed (again) by the modifier name, and ending with ".php". All this naming and positioning of code helps Smarty determine what file to include, and which function to invoke. So, whenever you use the '|substr' modifier in your template, Smarty knows to call this function.

The function code is so simple there's hardly anything to explain. As you can see, the modifier function simply calls the PHP standard **substr()** function, and returns the result of that function. By the way,

Listing 3: prefilter.literal_script.php

```
<?php

function smarty_prefilter_literal_script($source, &$smarty){
    $result=&$source;
    $result=preg_replace('~<script\b(?![^>]*smarty)~iU', '<!—{literal} —><script', $result);
    $result=preg_replace('~</script>~iU', '</script><!—{literal} {/literal}—>', $result);
    return $result;
}

?>
```

this modifier also performs in exactly the same manner as the PHP **substr()** function. The only difference is logistical; our modifier only accepts up to two parameters, rather than the possible three accepted by the **substr()** function. This is due to the first parameter in our modifier being implicitly added as the value that precedes the modifier. So, for example, **{$article_title|substr:0:3}** prints the first three characters of **$article_title**, whereas this would be written as **substr($article_title, 0, 3)** using native PHP. Likewise, **{$article_title|substr:-5}** behaves just like **substr($article_title, -5)**. That is, it prints the last 5 characters of a string.

Filters

Filters are functions that do some processing on the source of our templates either before or after they are compiled, which is why they're referred to as 'pre' and 'post' filters, respectively. Filters can also be invoked after the execution of a compiled template. These are called 'output filters', since they influence what is essentially the final output of the code. Filters are useful in various situations, but let's go over one in particular that you will run into the minute you start using Smarty in a real world project.

Smarty delimiters are { and }. Although they can be changed, they are still the most commonly used. Unfortunately, { and } characters are also used in every HTML page that uses JavaScript and/or CSS styles. When the Smarty parser comes across JavaScript or CSS code, it reports a compile error, because it doesn't know how to interpret the foreign code:

```
<STYLE type="text/css">
<!—
.article-title {  font-size: x-large; }
—>
</STYLE>
```

This is easily solved by wrapping your JavaScript or CSS code in the Smarty tags, **{literal}** and **{/literal}**. These tags tell Smarty that it shouldn't try to interpret the marked code. Since web designers tend to forget "details" like that, we need a more automated solution. This is accomplished with two pre-filters. In Listings 3 and 4, you can see a bit of regex magic that adds {literal} tags around **<SCRIPT>** and **<STYLE>** tags.

The explanation of regular expressions used in these functions is out of scope for this article, so if you don't understand that code, you will have to trust me that this pre-filter converts code from the above into something like:

```
<!— {literal} —><STYLE type="text/css">
<!—
.article-title  {
                  font-size: x-large;
                  }
—>
</STYLE><!—{literal} {/literal}—>
```

"But wait a minute!", you will say. "What if I *do* want to use Smarty variables inside script or style tags, or I want to let Smarty process contents inside these tags?". In that case, all you have to do is add a "smarty" pseudo-attribute to the opening HTML tag, like

Listing 4: prefilter.literal_style.php

```
<?php

function smarty_prefilter_literal_style($source, &$smarty){
    $result=&$source;
    $result=preg_replace('~<style\b(?![^>]*smarty)~iU', '<!—{literal} —><style', $result);
    $result=preg_replace('~</style>~iU', '</style><!—{literal} {/literal}—>', $result);
    return $result;
}

?>
```

Listing 5: outputfilter.literal_cleanup.php

```
<?php

function smarty_outputfilter_literal_cleanup($source, &$smarty){
    $result=&$source;
    $result=preg_replace('~<!—({literal})? —>~iU', '', $result);
    return $result;
}

?>
```

Listing 6: debug.inc.php

```
<?php

    session_start();
    if (isset($_GET['debug'])) {
        $_SESSION['debug']=$_GET['debug'];
    }
    $smarty_debugging=false;
    if (isset($_SESSION['debug'])) {
        $smarty_debugging=$_SESSION['debug'];
    }

    error_reporting(E_ALL);
    set_error_handler('error_handler');

function error_handler($type, $text, $file, $line,
                                                $vars)
{
    global $error_log;
    $error_log[]=$text;
    $error_log[]="[$line] $file";
}

function debug_log($key, $data) {
    global $debug_log;
    debug_log[$key][]=$data;
}

?>
```

this:

```
<SCRIPT language="JavaScript" smarty>
<!—
alert("Hello {$user_name}!");
//—>
</SCRIPT>
```

With this attribute, contents of script tag will be compiled into PHP code, and JavaScript will greet site visitors with an alert box containing their name.

These two pre-filters leave some junk behind them, in the form of empty and/or unnecessary HTML comment tags. I know that these HTML comments don't affect the final look of the page, but I have decided to implement an output filter that cleans this for two reasons. First, to test-drive how output-filters work, (since I haven't found a real, practical use for them yet), and second, to show you how easy it is to implement them.

First of all, what exactly are output filters? Well, if you've ever used output buffering in your PHP scripts, you will grasp the idea very quickly. When all of your PHP application code is executed, and when all of the templates are processed, before the HTML code is sent to the browser, it is processed with an "output handler", which modifies it if necessary. The output filter from Listing 5 does just that. It "cleans" HTML code from needless comment tags before sending the page to the browser. The code in the function is just one call to PHP's **`preg_replace()`** function, which does all of the dirty work for us.

All of these filters are explicitly loaded in the **`smarty()`** function in Listing 1 (lines 14, 15 and 16).

Debugging With Smarty

Smarty already has some built in mechanisms for debugging your templates. These methods can be easily extended and used for debugging your whole web application.

Smarty Debug Console, for example, is a popup window that informs you about all included templates and variables that were passed to Smarty. This can be very useful in detecting bugs in your templates. A pitfall in using this tool, however, is that whenever it is turned on it pops up in response to every page request made to your site. Fortunately, like everything else in Smarty, this behaviour can be changed too. Since debug console is just another template that comes with Smarty by default, it is easy to edit and alter its behaviour. I adjusted it to my taste, and the code can be found in the accompanying zip file.

In Listing 6 you can see a little trick that limits visibility of the debug console only to you. In line 4, variables passed via the HTTP GET method are checked. If the variable named "debug" is passed (with values 1 or 0, for on/off), its value is stored in a session variable "**`debug`**". Later, this value is used in the **`smarty()`** function (Listing 1, line 12) to show/hide the debug console. Smarty has a similar capability already built in, but my method is slightly better since you don't have to add a debug parameter in every URL you want to call.

This modification makes the debugging functionality within Smarty a bit less intrusive. When you need to debug your application, just visit a page on the site, adding "**`?debug=1`**" to the URL. After that, debug

mode is turned on just for you, and for every other visitor to your site, it is turned off. This remains in effect until you close your browser or add a "?debug=0" to the URL. This can be very handy for debugging live, production sites.

> *Note: this isn't the safest method for debugging, so you should probably turn this feature off manually whenever you don't need it.*

Later in Listing 6, you can see how to use the debug console for error handling. First, you turn all errors on with **`error_reporting()`**, and set **`error_handler()`** as your error handler. This function stores error messages in a global array **`$error_log`**, which is later passed to smarty in the function '**`smarty()`**' (Listing 1, line 18), so that you can see them later in the console.

> *Note: this is not the full-fledged, recommended error handling and logging procedure. You should still log errors to a file, a database or a syslog, or use one of the existing log classes for that job (look at PEAR::Log).*

Listing 6 contains another useful function called **`debug_log()`**. Its purpose is to log debugging messages to the debug console. It stores those messages to the **`$debug_log`** global array and can be used to provide meaningful error messages that PHP doesn't report as errors (like **`mysql_error()`**) or to an indication that your script did or didn't do something:

```
mysql_query($query);
if ($mysql_errno()) {
    debug_log('mysql_query', $query);
    debug_log('mysql_error', mysql_error());
}
```

Debug messages are stored in key->value pairs to increase readability. Later, the **`$debug_log`** variable is passed to Smarty in the **`smarty()`** function (Listing 1, line 19).

Bonus: Smarty MX

For extra lazy web designers (is that redundant?) who refuse to learn Smarty syntax, there is a solution that lets them use their favorite WYSIWYG editor (Macromedia Dreamweaver MX) and still leverage the power of PHP and Smarty.

Figure 1 shows all of the features of Dreamweaver templates. Two Editable, one Repeating and one

Figure 1: Dreamweaver MX Template Features

Figure 2: Dreamweaver template source code - smarty_mx.dwt

```
<!-- TemplateBeginRepeat name="items" -->

<tr>
  <td><!-- TemplateBeginEditable name="items.name" -->Apples<!-- TemplateEndEditable --></td>
  <td>$<!-- TemplateBeginEditable name="items.price" -->14.70<!-- TemplateEndEditable --></td>
  <td><!-- TemplateBeginIf cond="items.isnew" --><img src="new.gif"><!-- TemplateEndIf --></td>
</tr>

<!-- TemplateEndRepeat -->
```

Optional region are used. In Dreamweaver language, Editable regions are the only part of the page you are allowed to change in a file based on a template. In the same manner, you can control the visibility of Optional regions, and you can duplicate Repeating regions as many times as you like. These regions will perform slightly different functions when transfered to a Smarty template that we will produce from this template. We will replace Editable regions with regular template variables, Optional regions will form **if** statements, and Repeating regions will produce **foreach** statements.

This table was produced with the Repeating Table command (see the tool tip in the figure) so that the repeating region contains the whole table row, and will produce a new row for every item in the list.

Pay attention to the names of the regions in the light blue boxes above. We use these names to convert this Dreamweaver template into a Smarty template.

This Dreamweaver template source code can be seen in Figure 2, and the Smarty template code that we want to produce from this Dreamweaver template is shown in Figure 3. Note the region names we used in Dreamweaver. They are shown in 'name' pseudo-attributes in HTML comment tags. These comment tags are actually a method that Dreamweaver uses to mark template regions, similar to the Smarty tags in Figure 3. Note in fact, that the 'foreach' tag in Figure 3 uses the same name as the corresponding repeating

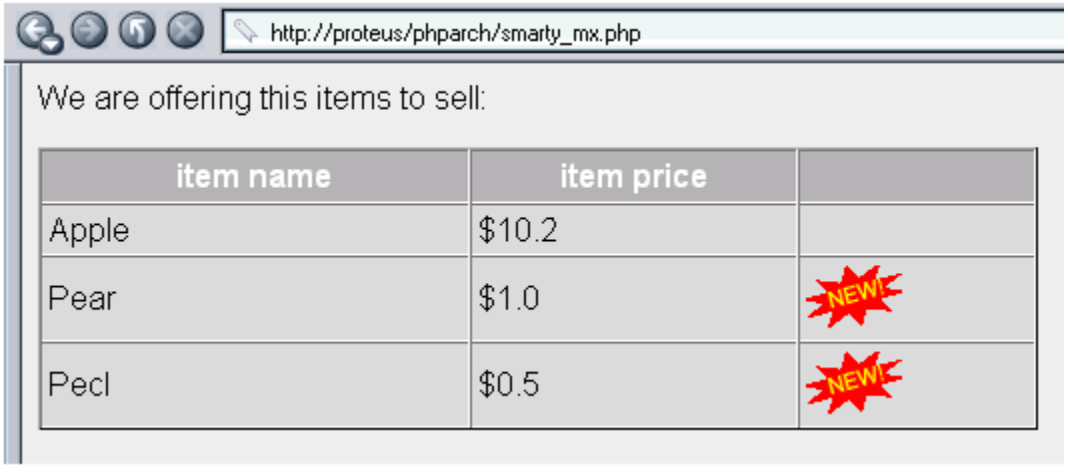
Listing 7: prefilter.dreamweaver_template.php

```
<?php

function smarty_prefilter_dreamweaver_template(&$source, &$smarty){
    $pattern[]='~<!— TemplateBeginEditable name="(.*)" —>.*<!— TemplateEndEditable —>~Us';
    $pattern[]='~<!— TemplateBeginIf cond="(.*)" —>(.*)<!— TemplateEndIf —>~Us';
    $pattern[]='~<!— TemplateBeginRepeat name="(.*)" —>(.*)<!— TemplateEndRepeat —>~Us';
    $replace[]='{$$1}';
    $replace[]='{if $$1}$2{/if}';
    $replace[]='{foreach from=$$1 item=$1}$2{/foreach}';
    return preg_replace($pattern, $replace, $source);
}

?>
```

Figure 3: Smarty template code for that template

```
{foreach from=$items item=$items}
<tr>
  <td>{$items.name}</td>
  <td>${$items.price}</td>
  <td>{if $items.isnew}<img src="new.gif">{/if}</td>
</tr>
{/foreach}
```

Figure 4: the results



Listing 8: example data for Dreamweaver template – smarty_mx.php

```
<?php

        include_once('debug.inc.php');
        include_once('smarty.inc.php');

        $user_name = 'Andrei Zmievski';

        items[] = array ('name'=>'Apple', 'price'=>'10.2', 'isnew'=>false);
        items[] = array ('name'=>'Pear', 'price'=>'1.0', 'isnew'=>true);
        items[] = array ('name'=>'Pecl', 'price'=>'0.5', 'isnew'=>true);

        $vars=compact('user_name', 'items');
        smarty('smarty_mx.dwt', $vars);

?>
```

region. This also holds true for simple template variables, and the condition in the 'if' smarty tag (the same as the one in optional region).

Listing 7 shows a pre-filter function that does the conversion from Dreamweaver template to a Smarty template. Since there are three types of regions, the **`$pattern`** array has three regular expressions, just like the **`$replace`** array. The filter uses only the simplest of regular expressions, yet does so much to ease the lives of PHP programmers and web designers.

Finally, Listing 8 contains sample code that generates data to fill this Dreamweaver-generated, Smarty-parsed template. The result can be seen in Figure 4.

At the End

It is really amazing how such a small amount of code can produce such a dramatic result. Actually, most of what we've done here isn't even possible using other template engines! And we did all this without needing to inspect any of the Smarty classes to decipher their inner workings. In the end, the code itself is clean and readable (or at least as readable as regular expressions can be).

These aren't even all of the features of Smarty. I haven't even made mention of caching, configuration files or resources (the ability to read template code from a database or any other source). I'll leave these topics as an exercise to the user (or at least another article).

I can only hope that I managed to bring you closer and give you an insight to advanced Smarty features and how they can be useful. If you haven't used Smarty yet, I hope that you are now able to recognize its potential as a useful website or web-based application development tool.

php|a

Tomica Jovanovic is a freelance programmer from Serbia, working his way trough college. You can contact him at tomica@mbox.co.yu