Ajax and Smarty, Part 1: Develop Ajax applications with Smarty

Build Ajax applications using PHP, the Smarty template engine, and the jQuery framework

Andrei Cioroianu

April 27, 2010

Smarty is a PHP template engine that lets you separate the business logic from the presentation in your Web applications. Smarty currently has no built-in Asynchronous JavaScript and XML (Ajax) support, but its plug-in architecture lets you extend it easily and use it together with JavaScript frameworks, such as jQuery. This series describes how to use Smarty in Ajax applications, how to create Smarty plug-ins, and how to improve the code quality of your Web applications, making the code more readable and easier to maintain.

View more content in this series

In this first article of the series, you'll learn how to use Smarty tempaltes to generate JSON, XML, and HTML responses for the Ajax requests. These techniques allow you to focus on the application logic when developing the PHP code, which is separated from the data formats that are used to communicate between the Ajax client and the server.

You'll also learn how to build a form that has two versions, one letting the user enter data in the input fields and the other using hidden fields and presenting the data in a non-editable manner. By clicking a button, the user can switch between the two versions of the form, submitting the data to the server with Ajax and retrieving the HTML content that is needed to update the page. In addition, the form would still work if JavaScript was disabled in the Web browser.

The last section of the article contains instructions for configuring Smarty and the sample application. This process is a bit complicated if your server or workstation has SELinux enabled. The extra commands are a small price to pay for the enhanced security provided by SELinux. You'll find this information useful for any Web application that has to modify its public files, such as content management systems and Web sites that allow their users to upload content. No matter if you use Smarty, a popular CMS, or a custom-built system, you'll run into the same configuration problems related to SELinux as soon as your code tries to modify the Web files. This article shows how to solve them, using the `restorecon`, `chcon`, and `setsebool` commands of Linux®.

Generating Ajax responses with Smarty

In this section, learn how to build Smarty templates that produce the responses for the Ajax requests. You can use any of the common formats, such as JSON, XML, or HTML. The Smarty syntax was designed primarily for HTML, which makes it very suitable for XML as well. However, it is a bit harder to build JSON responses with Smarty because template constructs use `{` and `}` in their syntax, which means you'll have to escape these two characters when they are used by JSON. You'll see, however, that it is possible to change the Smarty delimiters to avoid this syntax conflict. You'll also learn how to create custom modifiers and functions that are registered to the Smarty framework so that you can use them in your templates.

Producing XML documents with Smarty

Let's start with a simple example (shown in Listing 1) that generates an XML response that can be consumed by an Ajax client. First of all, the PHP code should set the content type and the `no-cache` headers, which ensure that the Web browser won't cache the Ajax responses. If you haven't used Smarty before, here is a quick explanation of what the demo_xml.php file is doing. It includes the Smarty class with `require`, creates a new instance of this class, sets its compilation and debugging flags, creates two variables named `root_attr` and `elem_data` with `assign()`, and then invokes a Smarty template named demo_xml.tpl with `display()`, which will generate the XML response.

Listing 1. The demo_xml.php example

```
<?php
header("Content-Type: text/xml");
header("Cache-Control: no-cache");
header("Pragma: no-cache");

require 'libs/Smarty.class.php';

$smarty = new Smarty;

$smarty->compile_check = true;
$smarty->debugging = false;
$smarty->force_compile = 1;

$smarty->assign("root_attr", "< abc & def >");
$smarty->assign('elem_data', array("111", "222", "333"));

$smarty->display('demo_xml.tpl');

?>
```

The demo_xml.tpl template (see Listing 2) produces a `<root>` element that has an attribute whose value is retrieved from the `root_attr` variable created in the demo_xml.php file. The `<`, `>`, `"`, `'` and `&` characters are replaced with `&lt;`, `&gt;`, `&quot;`, `&apos;`, and `&amp;`, respectively, using the `escape` modifier of Smarty. Within the `root` element, the template uses the `{section}` function of Smarty to iterate over the elements of the `elem_data` array, which is the second variable assigned in the demo_xml.php file. For each element of the array, the demo_xml.tpl template generates an XML element containing the value retrieved from the array.

Listing 2. The demo_xml.tpl template

```
<root attr="{$root_attr|escape}">
    {section name="d" loop=$elem_data}
        <elem>{$elem_data[d]|escape}</elem>
    {/section}
</root>
```

Listing 3 contains the XML output produced by the demo_xml.php file and the demo_xml.tpl template.

Listing 3. The XML output

```
<root attr="&lt; abc &amp; def &gt;">
            <elem>111</elem>
            <elem>222</elem>
            <elem>333</elem>
</root>
```

Building a JSON response with Smarty

The demo_json.php file (shown in Listing 4) sets the no-cache headers, and creates and configures the Smarty object just like the example in Listing 3. In addition, it defines two functions named `json_modifier()` and `json_function()`, which call `json_encode()` of PHP. The two functions are registered to Smarty so that they can be used in templates, as you'll see later in this section. After that, the demo_json.php file creates a few Smarty variables of various types: a string, a number, a boolean, and an array. Then, the PHP example executes the demo_json.tpl template to produce the JSON response.

Listing 4. The demo_json.php example

```
<?php
header("Content-Type: application/json");
header("Cache-Control: no-cache");
header("Pragma: no-cache");

require 'libs/Smarty.class.php';

$smarty = new Smarty;

$smarty->compile_check = true;
$smarty->debugging = false;
$smarty->force_compile = 1;

function json_modifier($value) {
    return json_encode($value);
}

function json_function($params, &$smarty) {
    return json_encode($params);
}

$smarty->register_modifier('json', 'json_modifier');
$smarty->register_function('json', 'json_function');

$smarty->assign('str', "a\"b\"c");
$smarty->assign('num', 123);
$smarty->assign('bool', false);
```

```
$smarty->assign('arr', array(1,2,3));

$smarty->display('demo_json.tpl');

?>
```

Instead of registering the plug-ins (such as modifiers or functions) to Smarty, you can follow some special naming convention described in the Smarty documentation (see Related topics). You can then place your code in a plug-ins directory so that your Smarty modifiers and functions can be used in any Web page of the application.

Because both JSON and Smarty use `{` and `}` in their syntax, you must use `{ldelim}` and `{rdelim}` in your Smarty templates to generate the `{` and `}` characters of the JSON response. You can also place `{` and `}` between `{literal}` and `{/literal}`. As you'll see in another example, it is possible to change the Smarty delimiters to avoid this inconvenience.

The `demo_json.tpl` template (see Listing 5) uses the `json` modifier to encode the values of the four variables that are set in the demo_json.php file. This is useful, for example, to escape the quotes and other special characters such as tabs and new lines in a string. Smarty will call the `json_modifier()` function from the demo_json.php file each time the template uses `|json`. The `json` modifier must be preceded by the `@` character so that the array variable can be passed to `json_modifier()`. Without the `@` character, the modifier function would be called for each element of the array.

The `{json ... }` construct from the demo_json.tpl template is translated to a call to `json_function()` of the demo_json.php file. The function gets the attributes from the template as an array named `params` that is passed to `json_encode()`, which returns a JSON representation of the PHP associative array.

Listing 5. The demo_json.tpl template

```
{ldelim}
 s: {$str|json},
 n: {$num|json},
 b: {$bool|json},
 a: {$arr|@json},
 o: {json os=$str on=$num ob=$bool oa=$arr},
 z: {literal}{ x: 1, y: 2 }{/literal}
{rdelim}
```

Listing 6 contains the JSON output produced by the demo_json.php file and the demo_json.tpl template.

Listing 6. The JSON output

```
{
 s: "a\"b\"c",
 n: 123,
 b: false,
 a: [1,2,3],
 o: {"os":"a\"b\"c","on":123,"ob":false,"oa":[1,2,3]},
 z: { x: 1, y: 2 }
}
```

To avoid the use of `{ldelim}` and `{rdelim}` in Smarty templates, you can change the Smarty delimiters as shown in Listing 7.

Listing 7. Changing the Smarty delimiters

```
$smarty->left_delimiter = '<%';
$smarty->right_delimiter = '%>';
```

The template from Listing 8 generates the JSON response, using the `<%` and `%>` delimiters in the Smarty constructs.

Listing 8. The demo_json2.tpl template

```
{
 s: <% $str|json %>,
 n: <% $num|json %>,
 b: <% $bool|json %>,
 a: <% $arr|@json %>,
 o: <% json os=$str on=$num ob=$bool oa=$arr %>,
 z: {  x: 1, y: 2 }
}
```

Creating an Ajax page with Smarty

The example from this section shows how to use Smarty to generate HTML content that is retrieved with Ajax. In addition, the Web page has an HTML form whose data is submitted with Ajax to the server, using the jQuery framework. If JavaScript is disabled in the Web browser, the form still works properly and Smarty is still used to produce the content on the server.

Using Smarty with HTML forms

The demo_form.tpl template (see Listing 9) contains an HTML form whose fields may be editable or not depending on the value of a variable named `edit_mode`. This variable is set in the PHP code invoking the template, as you'll see later in this section. The value of `edit_mode` is also stored in a hidden field of the form.

Listing 9. The HTML form of the demo_form.tpl template

```
<form method="POST" name="demo_form">

<input type="hidden" name="edit_mode"
    value="{if $edit_mode}true{else}false{/if}">

<table border="0" cellpadding="5" cellspacing="0">
    ...
</table>

</form>
```

Listing 10 shows the first field of the form, which is an input box if `edit_mode` is `true` or a hidden field if `edit_mode` is `false`. In the latter case, the non-editable value of the field is included in the output with `{$smarty.post.demo_text|escape}`. When the user submits the editable form, the `demo_text` parameter contains the user's input. When the form is not editable, the parameter is still present, thanks to the hidden field. Therefore, it is possible to obtain the value of the post parameter with `$smarty.post.demo_text` whether the form is editable or not.

Listing 10. The text field of the demo_form.tpl template

```
    <tr>
        <td>Demo Text:</td>
        <td>
            {if $edit_mode}
                <input type="text" name="demo_text" size="20"
                    value="{$smarty.post.demo_text|escape}">
            {else}
                <input type="hidden" name="demo_text"
                    value="{$smarty.post.demo_text|escape}">
                {$smarty.post.demo_text|escape}
            {/if}
        </td>
    </tr>
```

The next input field of the form is a checkbox (see Listing 11). In the editable version of the form, the `input` element has the `checked` attribute only if the `demo_checkbox` parameter is present. Similarly, the non-editable version of the form contains the hidden form element only if the submitted form data contains the post parameter named `demo_checkbox`.

Listing 11. The checkbox of the demo_form.tpl template

```
    <tr>
        <td>Demo Checkbox:</td>
        <td>
            {if $edit_mode}
                <input type="checkbox" name="demo_checkbox"
                    {if $smarty.post.demo_checkbox}checked{/if}>
            {else}
                {if $smarty.post.demo_checkbox}
                    <input type="hidden" name="demo_checkbox" value="On">
                {/if}
                {if $smarty.post.demo_checkbox}On{else}Off{/if}
            {/if}
        </td>
    </tr>
```

The following row of the form's table contains three radio buttons (see Listing 12). The template's code determines which radio button should be selected by comparing each button's value with the `demo_radio` parameter. The non-editable form uses a hidden input field to store the parameter's value and shows that value to the user with `$smarty.post.demo_radio`.

Listing 12. The radio buttons of the demo_form.tpl template

```
    <tr>
        <td>Demo Radio:</td>
        <td>
            {if $edit_mode}
                <input type="radio" name="demo_radio" value="1"
                    {if $smarty.post.demo_radio == '1'}checked{/if}>1
                <input type="radio" name="demo_radio" value="2"
                    {if $smarty.post.demo_radio == '2'}checked{/if}>2
                <input type="radio" name="demo_radio" value="3"
                    {if $smarty.post.demo_radio == '3'}checked{/if}>3
            {else}
                <input type="hidden" name="demo_radio"
                    value="{$smarty.post.demo_radio|escape}">
                {$smarty.post.demo_radio|escape}
            {/if}
        </td>
    </tr>
```

The options of a form list are generated in a loop with `{section}`, as shown in Listing 13. The current index of the loop is assigned to a template variable named `demo_counter`, which is compared with the value of the option element being generated to determine if the option should be selected.

Listing 13. The list of the demo_form.tpl template

```
    <tr>
        <td>Demo Select:</td>
        <td>
            {if $edit_mode}
                <select name="demo_select" size="1">
                    {section name="demo_section" start=10 loop=100 step="10"}
                        {assign var="demo_counter"
                            value=$smarty.section.demo_section.index}
                        <option {if $smarty.post.demo_select == $demo_counter}
                                selected{/if} value="{$demo_counter}">
                            {$demo_counter}
                        </option>
                    {/section}
                </select>
            {else}
                <input type="hidden" name="demo_select"
                    value="{$smarty.post.demo_select|escape}">
                {$smarty.post.demo_select|escape}
            {/if}
        </td>
    </tr>
```

The submit button has a different label (Save or Edit) depending on the value of the `edit_mode` flag (see Listing 14). The `onclick` attribute contains a call to a JavaScript function named `submitDemoForm()`. As you'll see later in the article, this function uses Ajax to submit the form's data to the server and returns `false` so the Web browser doesn't submit the same data one more time in response to the button's click. If JavaScript is disabled, however, `submitDemoForm()` won't be called, and the Web browser does submit the form to the server. Therefore, the form will work properly no matter if JavaScript is enabled or disabled.

Listing 14. The submit button of the demo_form.tpl template

```
    <tr>
        <td>&nbsp;</td>
        <td>
            <button type="submit" onclick="return submitDemoForm()">
                {if $edit_mode}Save{else}Edit{/if}
            </button>
        </td>
    </tr>
```

Developing the page template

The demo_page.tpl file (see Listing 15) contains two `<script>` elements, one for jQuery and the other for the JavaScript file of the sample application. The page template includes the form template within a `<div>` element, using the `{include}` function of Smarty.

Listing 15. The demo_page.tpl template

```
<html>
<head>
    <title>Demo</title>
    <script type="text/javascript"
        src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js">
    </script>
    <script type="text/javascript" src="demo_js.js">
    </script>
</head>
<body>
    <div id="demo_div">
        {include file="demo_form.tpl"}
    </div>
</body>
</html>
```

Building a PHP controller for Smarty and Ajax

The demo_html.php file (shown in Listing 16) acts as a bridge between Ajax and Smarty, handling the Ajax requests and using Smarty to generate the Ajax responses with the demo_form.tpl template, which is invoked only if an Ajax header is present. This header is set in the JavaScript code, as you'll see in the following sub-section. The code uses the demo_page.tpl template if the Ajax header is missing, which means that either this is the initial page request made by the Web browser or JavaScript is disabled. After each request the value of the `edit_mode` flag is inverted to alternate the editable form and its non-editable version.

Listing 16. The demo_html.php example

```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");

require 'libs/Smarty.class.php';

$smarty = new Smarty;

$smarty->compile_check = true;
$smarty->debugging = false;
$smarty->force_compile = 1;

$edit_mode = @($_REQUEST['edit_mode'] == "true");
$smarty->assign("edit_mode", !$edit_mode);

$ajax_request = @($_SERVER["HTTP_AJAX_REQUEST"] == "true");
$smarty->display($ajax_request ? 'demo_form.tpl' : 'demo_page.tpl');

?>
```

Using Ajax to invoke Smarty templates

The `submitDemoForm()` function (see Listing 17) is called when the form's button is clicked. It sends the form's data to the server with jQuery, using the `POST` method and the same URL used by the Web form. The form's data is encoded as a string using the `serialize()` API of jQuery. The `beforeSend()` function, which is called by jQuery before sending the Ajax request, is used in this example to set the `Ajax-Request` header that is needed on the server side to identify the

Ajax requests. The `success()` function is called when the Ajax response is received. This callback inserts the response's content into the `<div>` element of the Web page.

Listing 17. The demo_js.js example

```
function submitDemoForm() {
    var form = $("form[name=demo_form]");
    $.ajax({
        type: "POST",
        url: form.action ? form.action : document.URL,
        data: $(form).serialize(),
        dataType: "text",
        beforeSend: function(xhr) {
            xhr.setRequestHeader("Ajax-Request", "true");
        },
        success: function(response) {
            $("#demo_div").html(response);
        }
    });
    return false;
}
```

Set up Smarty with SELinux enabled

After unzipping the sample application, you should see a directory named ajaxsmarty that contains the PHP files, a JavaScript file, and fours sub-directories: cache, configs, templates and templates_c. The templates directory contains the Smarty templates of the sample application. The other three sub-directories are empty.

Download the latest stable release of Smarty (see Related topics) and unzip it. (The sample application was tested with Smarty 2.6.25.) Next, copy the libs sub-directory of Smarty into the ajaxsmarty directory, which is the main directory of the sample application.

Upload or copy the ajaxsmarty directory (containing both the sample application and Smarty's `libs`) into the html directory of Apache. If you're using a Web hosting company for your server, SELinux may be disabled because it would probably generate too many support calls. If you are testing the application on your own Linux installation, chances are SELinux is enabled, and you might get the following error when requesting a PHP file from your browser: "SELinux is preventing the httpd from using potentially mislabeled files." The solution is to run the command from Listing 18 as root.

Listing 18. Setting the security context (labels) of the Web files

```
restorecon -R -v /var/www/html/ajaxsmarty
```

At this point, you should be able to open into your browser http://localhost/ajaxsmarty/, which shows three links. If you click one of them you'll get the following Smarty error in the Web browser: "Fatal error: Smarty error: unable to write to $compile_dir '/var/www/html/ajaxsmarty/templates_c'. Be sure $compile_dir is writable by the Web server user. in /var/www/html/ajaxsmarty/libs/Smarty.class.php on line 1113"

The above error happens because the Smarty setup is not yet complete. You must give the Web server user permissions to write into the templates_c and cache directories. The right way to do that is to change their owner as shown in Listing 19. Note that the `apache` user name and the server's html directory might be different on your computer.

Listing 19. Changing the owner of two directories so that Smarty can create its files

```
chown apache:apache /var/www/html/ajaxsmarty/templates_c
chown apache:apache /var/www/html/ajaxsmarty/cache
```

If you don't have root access, you could change the write permissions of templates_c and cache instead of using `chown`. You should be able to do that using your FTP client, or you can use the `chmod` command with the 777 parameter. Allowing any user to write into those folders is not a very good idea, but it might be your only immediate option if you cannot use `chown`. If your Web server is public, you should contact the server administrator.

If SELinux is enabled on your computer, you might still get one of the following errors in the browser: "SELinux prevented httpd reading and writing access to http files." or "SELinux is preventing httpd (httpd_t) write to ./templates_c (public_content_rw_t)." The solution is to run the commands from Listing 20 as root.

Listing 20. Allowing Smarty to create files in its directories when SELinux is enabled

```
chcon -t public_content_rw_t /var/www/html/ajaxsmarty/templates_c
chcon -t public_content_rw_t /var/www/html/ajaxsmarty/cache
setsebool -P allow_httpd_anon_write=1
```

The `setsebool` command with the `allow_httpd_anon_write` parameter must be executed only once. It allows the httpd daemon to write files in the directories labeled `public_content_rw_t`.

Conclusion

In this article, you learned how to build Smarty templates that produce JSON, XML, and HTML responses for the Ajax requests, how to use Smarty to build an Ajax form that still works if JavaScript is disabled in the Web browser, and how to configure Smarty on Linux machines that have SELinux enabled.

Downloadable resources

Description	Name	Size
Sample application for this article	ajaxsmarty_part1_src.zip	5KB

Related topics

- The Smarty manual contains everything you need to know about Smarty.
- Take a look at this sample application based on Smarty.
- "Separate form and function in PHP applications with Smarty" (developerWorks, July 2007) is an introduction to Smarty provided by developerWorks.
- The jQuery framework complements Smarty, adding the needed JavaScript capabilities, such as Ajax.
- The developerWorks Ajax resource center contains a growing library of Ajax content as well as useful resources to get you started developing Ajax applications today.
- Download Smarty and start using it today.