SmartyBundle Documentation

Release 0.2.0

Vítor Brandão

January 26, 2013

CONTENTS

This Symfony2 bundle provides integration for the Smarty3 template engine.

Contents

CHAPTER

ONE

INTRODUCTION

This bundle was created to support Smarty in Symfony2, providing an alternative to the Twig template engine natively supported.

> An effort was made to provide, where possible, the same user configuration and extensions available for the Twig bundle. This is to allow easy switching between the two bundles (at least I hope so!).

1.1 What is Smarty?

Smarty is a template engine for PHP, facilitating the separation of presentation (HTML/CSS) from application logic. This implies that PHP code is application logic, and is separated from the presentation.

Some of Smarty's features: [1]

- It is extremely fast.
- It is efficient since the PHP parser does the dirty work.
- No template parsing overhead, only compiles once.
- It is smart about recompiling only the template files that have changed.
- You can easily create your own custom functions and variable modifiers, so the template language is extremely extensible.
- Configurable template `{delimiter}` tag syntax, so you can use `{$foo}`, `{{$foo}}`, `<!--{$foo}-->`, etc.
- The `{if}..{elseif}..{else}..{/if}` constructs are passed to the PHP parser, so the `{if...}` expression syntax can be as simple or as complex an evaluation as you like.
- Allows unlimited nesting of sections, if's etc.
- Built-in caching support.
- Arbitrary template sources.
- Template Inheritance for easy management of template content.
- Plugin architecture.

See the Smarty3 Manual for other features and information on it's syntax, configuration and installation.

[1] http://www.smarty.net/docs/en/what.is.smarty.tpl

CHAPTER

TWO

REQUIREMENTS

- PHP 5.3.2 and up.
- Smarty 3
- Symfony 2

CHAPTER

THREE

LICENSE

This bundle is licensed under the LGPLv3 License. See the LICENSE file for details.

CHAPTER

FOUR

INSTALLATION

4.1 Download SmartyBundle

This can be done in several ways, depending on your preference. The first method is the standard Symfony2.1 method.

Using composer:

Add SmartyBundle in your `composer.json`:

```
{
    "require": {
        "noiselabs/smarty-bundle": "dev-master"
    }
}
```

Now tell composer to download the bundle by running the command:

```
$ php composer.phar update noiselabs/smarty-bundle
```

Composer will install the bundle to your project's *vendor/noiselabs* directory.

Using the vendors script:

Add the following lines in your `deps` file::

```
[NoiseLabsSmartyBundle]
    git=git://github.com/noiselabs/SmartyBundle.git
    target=bundles/NoiseLabs/Bundle/SmartyBundle
```

Now, run the vendors script to download the bundle:

```
$ php bin/vendors install
```

Using submodules:

If you prefer instead to use git submodules, then run the following:

```
$ git submodule add git://github.com/noiselabs/SmartyBundle.git vendor/bundles/NoiseLabs/Bundle/Smart
$ git submodule update --init
```

4.2 Configure the Autoloader (only if not using composer!)

Add the `NoiseLabs` namespace to your autoloader:

```
<?php
// app/autoload.php

$loader->registerNamespaces(array(
    // ...
    'NoiseLabs\\Bundle' => __DIR__.'/../vendor/bundles',
));
```

4.3 Enable the bundle

Enable the bundle in the kernel:

```
<?php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new NoiseLabs\Bundle\SmartyBundle\SmartyBundle(),
    );
}
```

4.4 Enable the Smarty template engine in the config

```
# app/config/config.yml
# ...
framework:
    templating:      { engines: ['twig', 'smarty'] }
```

CHAPTER

FIVE

USAGE

5.1 Basic usage

You can render a Smarty template instead of a Twig one simply by using the **.smarty** extension in the template name instead of .twig. The controller below renders the index.html.smarty template:

```
// src/Acme/HelloBundle/Controller/HelloController.php

public function indexAction($name)
{
    return $this->render('AcmeHelloBundle:Hello:index.html.smarty', array('name' => $name));
}
```

5.2 Template Inheritance

Like Symfony2 PHP renderer or Twig, Smarty provides template inheritance.

> Template inheritance is an approach to managing templates that resembles object-oriented programming techniques. Instead of the traditional use of `{include ...}` tags to manage parts of templates, you can inherit the contents of one template to another (like extending a class) and change blocks of content therein (like overriding methods of a class.) This keeps template management minimal and efficient, since each template only contains the differences from the template it extends.

Example:

layout.html.smarty:

```
<html>
<head>
    <title>{block name=title}Default Page Title{/block}</title>
</head>
<body>
    {block name=body}{/block}
</body>
</html>
```

mypage.html.smarty:

```
{extends 'file:AcmeHelloBundle:Default:layout.html.smarty'}
{block name=title}My Page Title{/block}
{block name=body}My HTML Page Body goes here{/block}
```

Output of mypage.html.smarty:

```
<html>
<head>
    <title>My Page Title</title>
</head>
<body>
    My HTML Page Body goes here
</body>
</html>
```

Instead of using the `file:AcmeHelloBundle:Default:layout.html.smarty` syntax you may use `file:[WebkitBundle]/Default/layout.html.smarty` which should be, performance wise, slightly better/faster (since this is a native Smarty syntax).:

```
{extends 'file:[WebkitBundle]/Default/layout.html.smarty'}
```

To load a template that lives in the `app/Resources/views` directory of the project you should use the following syntax:

```
{extends 'file:base.html.smarty'}
```

Please see Symfony2 - Template Naming and Locations to learn more about the naming scheme and template locations supported in Symfony2.

{include} functions work the same way as the examples above.:

```
{include 'file:WebkitBundle:Default:layout.html.smarty'}
{include 'file:[WebkitBundle]/Default/layout.html.smarty'}
{include 'file:base.html.smarty'}
```

Important: Note the usage of the `file:` resource in the `{extends}` function. We need to declare the resource even if the Smarty class variable `$default_resource_type` is set to `'file'`. This is required because we need to trigger a function to handle 'logical' file names (only mandatory if you are using the first syntax). Learn more about resources in the Smarty Resources webpage.

> The *.html.smarty* extension can simply be replaced by *.smarty*. We are prefixing with *.html* to stick with the Symfony convention of defining the format (*.html*) and engine (*.smarty*) for each template.

5.3 Injecting variables into all templates (i.e. Global Variables)

As exemplified in the Symfony Cookbook it is possible to make a variable to be accessible to all the templates you use by configuring your *app/config/config.yml* file:

```
# app/config/config.yml
smarty:
    # ...
    globals:
        ga_tracking: UA-xxxxx-x
```

Now, the variable ga_tracking is available in all Smarty templates:

```
<p>Our google tracking code is: {$ga_tracking} </p>
```

CHAPTER

SIX

EXTENSIONS

Smarty[Bundle] extensions are packages that add new features to Smarty. The extension architecture implemented in the SmartyBundle is an object-oriented approach to the plugin system available in Smarty. The implemented architecture was inspired by Twig Extensions.

Each extension object share a common interest (translation, routing, etc.) and provide methods that will be registered as a Smarty plugin before rendering a template. To learn about the plugin ecosystem in Smarty take a look at the Smarty documentation page on that subject.

The SmartyBundle comes with a few extensions to help you right away. These are described in the next section.

6.1 Actions Extension

This extension tries to provide the same funcionality described in Symfony2 - Templating - Embedding Controllers.

Following the example presented in the link above, the Smarty equivalents are:

Using a block function:

{render attributes=['min'=>1,'max'=>3]}AcmeArticleBundle:Article:recentArticles{/render}

Using a modifier:

{'AcmeArticleBundle:Article:recentArticles'|render:['min'=>1,'max'=>3]}

6.2 Assetic Extension

Assetic is an asset management framework for PHP. This extensions provides support for it's usage in Symfony2 when using Smarty templates.

Assetic combines two major ideas: assets and filters. The assets are files such as CSS, JavaScript and image files. The filters are things that can be applied to these files before they are served to the browser. This allows a separation between the asset files stored in the application and the files actually presented to the user.

Using Assetic provides many advantages over directly serving the files. The files do not need to be stored where they are served from and can be drawn from various sources such as from within a bundle:

```
{javascripts
    assets='@AcmeFooBundle/Resources/public/js/*'
}
<script type="text/javascript" src="{$asset_url}"></script>
{/javascripts}
```

To bring in CSS stylesheets, you can use the same methodologies seen in this entry, except with the stylesheets tag:

```
{stylesheets
    assets='@AcmeFooBundle/Resources/public/css/*'
}
<link rel="stylesheet" href="{$asset_url}" />
{/stylesheets}
```

6.2.1 Combining Assets

You can also combine several files into one. This helps to reduce the number of HTTP requests, which is great for front end performance. It also allows you to maintain the files more easily by splitting them into manageable parts. This can help with re-usability as you can easily split project-specific files from those which can be used in other applications, but still serve them as a single file:

```
{javascripts
    assets='@AcmeFooBundle/Resources/public/js/*,
            @AcmeBarBundle/Resources/public/js/form.js,
            @AcmeBarBundle/Resources/public/js/calendar.js'
}
<script src="{$asset_url}"></script>
{/javascripts}
```

In the dev environment, each file is still served individually, so that you can debug problems more easily. However, in the prod environment, this will be rendered as a single script tag.

6.2.2 Block attributes

Here is a list of the possible attributes to define in the block function.

- `assets`: A comma-separated list of files to include in the build (CSS, JS or image files)
- `debug`: If set to true, the plugin will not combine your assets to allow easier debug
- `filter`: A coma-separated list of filters to apply. Currently, only LESS and YuiCompressor (both CSS and JS) are supported
- `combine`: Combine all of your CSS and JS files (overrides *debug*)
- `output`: Defines the URLs that Assetic produces
- `var_name`: The variable name that will be used to pass the asset URL to the <link> tag
- `as`: An alias to `var_name`. Example: `as='js_url'`
- `vars`: Array of asset variables. For a description of this recently added feature please check out the Johannes Schmitt blog post about Asset Variables in Assetic.

  Note: Unlike the examples given in the Asset Variables in Assetic, which uses curly brackets for the `vars` placeholder we are using **square brackets** due to Smarty usage of curly brackets as syntax delimiters. So `js/messages.{locale}.js` becomes `js/messages.[locale].js`.

6.2.3 Full example

Example using all available attributes:

```
{javascripts
    assets='@AcmeFooBundle/Resources/public/js/*,
            @AcmeBarBundle/Resources/public/js/form.js,
            @AcmeBarBundle/Resources/public/js/calendar.js',
            @AcmeBarBundle/Resources/public/js/messages.[locale].js
    filter='yui_js'
    output='js/compiled/main.js'
    var_name='js_url'
    vars=['locale']
}
<script src="{$js_url}"></script>
{/javascripts}
```

6.2.4 Symfony/Assetic documentation

For further details please refer to the Symfony documentation pages about Assetic:

- How to Use Assetic for Asset Management
- How to Minify JavaScripts and Stylesheets with YUI Compressor

6.3 Assets Extension

Templates commonly refer to images, Javascript and stylesheets as assets. You could hard-code the path to these assets (e.g. `/images/logo.png`), but the SmartyBundle provides a more dynamic option via the `assets` function:

```
<img src="{asset}images/logo.png{/asset}" />

<link href="{asset}css/blog.css{/asset}" rel="stylesheet" type="text/css" />
```

This bundle also provides the `assets_version` function to return the version of the assets in a package. To set the version see the assets_version configuration option in Symfony's Framework Bundle.

Usage in template context:

```
{assets_version}
```

6.4 Form Extension

Coming soon.

6.5 Routing Extension

To generate URLs from a Smarty template you may use two block functions (`path` and `url`) provided by the RoutingExtension.

```
<a href="{path slug='my-blog-post'}blog_show{/path}">
    Read this blog post.
</a>
```

Absolute URLs can also be generated.:

```
<a href="{url slug='my-blog-post'}blog_show{/url}">
    Read this blog post.
</a>
```

Please see the Symfony2 - Routing for full information about routing features and options in Symfony2.

6.6 Translation Extension

To help with message translation of static blocks of text in template context, the SmartyBundle, provides a translation extension. This extension is implemented in the class TranslationExtension.

You may translate a message, in a template, using a block or modifier. Both methods support the following arguments:

- **count**: In pluralization context, used to determine which translation to use and also to populate the %count% placeholder *(only available in transchoice)*;
- **vars**: Message placeholders;
- **domain**: Message domain, an optional way to organize messages into groups;
- **locale**: The locale that the translations are for (e.g. en_GB, en, etc);

`trans` block:

```
{trans}Hello World!{/trans}

{trans vars=['%name%' => 'World']}Hello %name%{/trans}

{trans domain="messages" locale="pt_PT"}Hello World!{/trans}

<!-- In case you're curious, the latter returns "Olá Mundo!" :) -->
```

`trans` modifier:

```
{"Hello World!"|trans}

{"Hello %name%"|trans:['%name%' => 'World']}

{"Hello World!"|trans:[]:"messages":"pt_PT"}
```

Message pluralization can be achieved using `transchoice`:

> **Note:** Unlike the examples given in the Symfony documentation, which uses curly brackets for explicit interval pluralization we are using **square brackets** due to Smarty usage of curly brackets as syntax delimiters. So `{0} There is no apples` becomes `[0] There is no apples`.

`transchoice` block:

```
{transchoice count=$count}[0] There is no apples|[1] There is one apple|]1,Inf] There is %count% appl
```

`transchoice` modifier:

```
{'[0] There is no apples|[1] There is one apple|]1,Inf] There is %count% apples'|transchoice:$count}
<!-- Should write: "There is 5 apples" -->

The transchoice block/modifier automatically gets the %count% variable from the current context and p
```

6.7 Security Extension

This extension provides access control inside a Smarty template. This part of the security process is called authorization, and it means that the system is checking to see if you have privileges to perform a certain action. For full details about the Symfony2 security system check it's documentation page.

> If you want to check if the current user has a role inside a template, use the built-in `is_granted` modifier.

Usage:

```
{if 'IS_AUTHENTICATED_FULLY'|is_granted:$object:$field}
    access granted
{else}
    access denied
{/if}
```

6.8 Enabling custom Extensions

To enable a Smarty extension, add it as a regular service in one of your configuration, and tag it with `smarty.extension`. The creation of the extension itself is described in the next section.

YAML example:

```
services:
    smarty.extension.your_extension_name:
        class: Fully\Qualified\Extension\Class\Name
        arguments: [@service]
        tags:
            - { name: smarty.extension }
```

6.9 Creating a SmartyBundle Extension

Note: In version 0.1.0 class AbstractExtension was simply named Extension. Please update your code when migrating to 0.2.0.

An extension is a class that implements the ExtensionInterface. To make your life easier an abstract AbstractExtension class is provided, so you can inherit from it instead of implementing the interface. That way, you just need to implement the getName() method as the `Extension` class provides empty implementations for all other methods.

The `getName()` method must return a unique identifier for your extension:

```
<?php

namespace NoiseLabs\Bundle\SmartyBundle\Extension;

class TranslationExtension extends AbstractExtension
{
    public function getName()
    {
        return 'translator';
    }
}
```

Plugins

Plugins can be registered in an extension via the `getPlugins()` method. Each element in the array returned by `getPlugins()` must implement PluginInterface.

For each Plugin object three parameters are required. The plugin name comes in the first parameter and should be unique for each plugin type. Second parameter is an object of type `ExtensionInterface` and third parameter is the name of the method in the extension object used to perform the plugin action.

Please check available method parameters and plugin types in the Extending Smarty With Plugins webpage.

```
<?php

namespace NoiseLabs\Bundle\SmartyBundle\Extension;

use NoiseLabs\Bundle\SmartyBundle\Extension\Plugin\BlockPlugin;

class TranslationExtension extends Extension
{
    public function getPlugins()
    {
        return array(
            new BlockPlugin('trans', $this, 'blockTrans'),
        );
    }

    public function blockTrans(array $params = array(), $message = null, $template, &$repeat)
    {
        $params = array_merge(array(
            'arguments' => array(),
            'domain'    => 'messages',
            'locale'    => null,
        ), $params);

        return $this->translator->trans($message, $params['arguments'], $params['domain'], $params['l
    }
}
```

Filters

Filters can be registered in an extension via the `getFilters()` method.

Each element in the array returned by `getFilters()` must implement FilterInterface.

```
<?php

namespace NoiseLabs\Bundle\SmartyBundle\Extension;

use NoiseLabs\Bundle\SmartyBundle\Extension\Filter\PreFilter;

class BeautifyExtension extends Extension
{
    public function getFilters()
    {
        return array(
            new PreFilter($this, 'htmlTagsTolower'),
        );
    }

    // Convert html tags to be lowercase
    public function htmlTagsTolower($source, \Smarty_Internal_Template $template)
```

```
    {
        return preg_replace('!<(\w+)[^>]+>!e', 'strtolower("$1")', $source);
    }
}
```

Globals

Global variables can be registered in an extension via the `getGlobals()` method.

There are no restrictions about the type of the array elements returned by `getGlobals()`.

```
<?php

namespace NoiseLabs\Bundle\SmartyBundle\Extension;

class GoogleExtension extends Extension
{
    public function getGlobals()
    {
        return array(
            'ga_tracking' => 'UA-xxxxx-x'
        );
    }
}
```

CHAPTER

SEVEN

CONFIGURATION REFERENCE

The example below uses YAML format. Please adapt the example if using XML or PHP.

app/config/config.yml:

```
smarty:

    options:

        # See http://www.smarty.net/docs/en/api.variables.tpl
        allow_php_templates:
        allow_php_templates:
        auto_literal:
        autoload_filters:
        cache_dir:                    %kernel.cache_dir%/smarty/cache
        cache_id:
        cache_lifetime:
        cache_locking:
        cache_modified_check:
        caching:
        caching_type:
        compile_check:
        compile_dir:                  %kernel.cache_dir%/smarty/templates_c
        compile_id:
        compile_locking:
        compiler_class:
        config_booleanize:
        config_dir:                   %kernel.root_dir%/config/smarty
        config_overwrite:
        config_read_hidden:
        debug_tpl:
        debugging:
        debugging_ctrl:
        default_config_type:
        default_modifiers:
        default_resource_type:        file
        default_config_handler_func:
        default_template_handler_func:
        direct_access_security:
        error_reporting:
        escape_html:
        force_cache:
        force_compile:
        locking_timeout:
        merge_compiled_includes:
```

```
        php_handling:
        plugins_dir:
        smarty_debug_id:
        template_dir:                   %kernel.root_dir%/Resources/views
        trusted_dir:
        use_include_path:               false
        use_sub_dirs:                   true

    globals:

        # Examples:
        foo:                    "@bar"
        pi:                     3.14
```

7.1 Available options

allow_php_templates By default the PHP template file resource is disabled. Setting $allow_php_templates to TRUE will enable PHP template files.

auto_literal The Smarty delimiter tags `{` and `}` will be ignored so long as they are surrounded by white space. This behavior can be disabled by setting auto_literal to false.

autoload_filters If there are some filters that you wish to load on every template invocation, you can specify them using this variable and Smarty will automatically load them for you. The variable is an associative array where keys are filter types and values are arrays of the filter names.

cache_dir This is the name of the directory where template caches are stored. By default this is `%kernel.cache_dir%/smarty/cache`. **This directory must be writeable by the web server**.

cache_id Persistent cache_id identifier. As an alternative to passing the same `$cache_id` to each and every function call, you can set this `$cache_id` and it will be used implicitly thereafter. With a `$cache_id` you can have multiple cache files for a single call to `display()` or `fetch()` depending for example from different content of the same template.

cache_lifetime This is the length of time in seconds that a template cache is valid. Once this time has expired, the cache will be regenerated. See the page Smarty Class Variables - $cache_lifetime for more details.

cache_locking Cache locking avoids concurrent cache generation. This means resource intensive pages can be generated only once, even if they've been requested multiple times in the same moment. Cache locking is disabled by default.

cache_modified_check If set to `TRUE`, Smarty will respect the If-Modified-Since header sent from the client. If the cached file timestamp has not changed since the last visit, then a '304: Not Modified' header will be sent instead of the content. This works only on cached content without `{insert}` tags.

caching This tells Smarty whether or not to cache the output of the templates to the `$cache_dir`. By default this is set to the constant `Smarty::CACHING_OFF`. If your templates consistently generate the same content, it is advisable to turn on `$caching`, as this may result in significant performance gains.

caching_type This property specifies the name of the caching handler to use. It defaults to file, enabling the internal filesystem based cache handler.

compile_check Upon each invocation of the PHP application, Smarty tests to see if the current template has changed (different timestamp) since the last time it was compiled. If it has changed, it recompiles that template. If the template has yet not been compiled at all, it will compile regardless of this setting. By default this variable is set to `TRUE`. Once an application is put into production (ie the templates won't be changing), the compile check step is no longer needed. Be sure to set $compile_check to `FALSE` for maximum performance. Note that if you change this to `FALSE` and a template file is changed, you will *not* see the change since the template will not get

recompiled. If $caching is enabled and $compile_check is enabled, then the cache files will get regenerated if an involved template file or config file was updated. As of Smarty 3.1 `$compile_check` can be set to the value `Smarty::COMPILECHECK_CACHEMISS`. This enables Smarty to revalidate the compiled template, once a cache file is

regenerated. So if there was a cached template, but it's expired, Smarty will run a single compile_check before regenerating the cache.

compile_dir This is the name of the directory where compiled templates are located. By default this is `%kernel.cache_dir%/smarty/templates_c`. **This directory must be writeable by the web server**.

compile_id Persistant compile identifier. As an alternative to passing the same `$compile_id` to each and every function call, you can set this $compile_id and it will be used implicitly thereafter. With a `$compile_id` you can work around the limitation that you cannot use the same `$compile_dir` for different `$template_dirs`. If you set a distinct `$compile_id` for each `$template_dir` then Smarty can tell the compiled templates apart by their `$compile_id`. If you have for example a prefilter that localizes your templates (that is: translates language dependend parts) at compile time, then you could use the current language as `$compile_id` and you will get a set of compiled templates for each language you use. Another application would be to use the same compile directory across multiple domains / multiple virtual hosts.

compile_locking Compile locking avoids concurrent compilation of the same template. Compile locking is enabled by default.

compiler_class Specifies the name of the compiler class that Smarty will use to compile the templates. The default is 'Smarty_Compiler'. For advanced users only.

config_booleanize If set to `TRUE`, config files values of `on/true/yes` and `off/false/no` get converted to boolean values automatically. This way you can use the values in the template like so: `{if #foobar#}...{/if}`. If `foobar` was `on`, `true` or `yes`, the `{if}` statement will execute. Defaults to `TRUE`.

config_dir This is the directory used to store config files used in the templates. Default is `%kernel.root_dir%/config/smarty`.

config_overwrite If set to `TRUE`, the default then variables read in from config files will overwrite each other. Otherwise, the variables will be pushed onto an array. This is helpful if you want to store arrays of data in config files, just list each element multiple times.

config_read_hidden If set to `TRUE`, hidden sections ie section names beginning with a period(.) in config files can be read from templates. Typically you would leave this `FALSE`, that way you can store sensitive data in the config files such as database parameters and not worry about the template loading them. `FALSE` by default.

debug_tpl This is the name of the template file used for the debugging console. By default, it is named `debug.tpl` and is located in the `SMARTY_DIR`.

debugging This enables the debugging console. The console is a javascript popup window that informs you of the included templates, variables assigned from php and config file variables for the current script. It does not show variables assigned within a template with the `{assign}` function.

debugging_ctrl This allows alternate ways to enable debugging. `NONE` means no alternate methods are allowed. `URL` means when the keyword `SMARTY_DEBUG` is found in the `QUERY_STRING`, debugging is enabled for that invocation of the script. If `$debugging` is `TRUE`, this value is ignored.

default_config_type This tells smarty what resource type to use for config files. The default value is `file`, meaning that `$smarty->configLoad('test.conf')` and `$smarty->configLoad('file:test.conf')` are identical in meaning.

default_modifiers This is an array of modifiers to implicitly apply to every variable in a template. For example, to HTML-escape every variable by default, use `array('escape:"htmlall"')`. To make a variable exempt from default modifiers, add the 'nofilter' attribute to the output tag such as `{$var nofilter}`.

default_resource_type This tells smarty what resource type to use implicitly. The default value is file, meaning that `{include 'index.tpl'}` and `{include 'file:index.tpl'}` are identical in meaning.

default_config_handler_func This function is called when a config file cannot be obtained from its resource.

default_template_handler_func This function is called when a template cannot be obtained from its resource.

direct_access_security Direct access security inhibits direct browser access to compiled or cached template files. Direct access security is enabled by default.

error_reporting When this value is set to a non-null-value it's value is used as php's error_reporting level inside of `display()` and `fetch()`.

escape_html Setting `$escape_html` to `TRUE` will escape all template variable output by wrapping it in `htmlspecialchars({$output}, ENT_QUOTES, SMARTY_RESOURCE_CHAR_SET);`, which is the same as `{$variable|escape:"html"}`. Template designers can choose to selectively disable this feature by adding the `nofilter` flag: `{$variable nofilter}`. This is a compile time option. If you change the setting you must make sure that the templates get recompiled.

force_cache This forces Smarty to (re)cache templates on every invocation. It does not override the `$caching` level, but merely pretends the template has never been cached before.

force_compile This forces Smarty to (re)compile templates on every invocation. This setting overrides `$compile_check`. By default this is `FALSE`. This is handy for development and debugging. It should never be used in a production environment. If `$caching` is enabled, the cache file(s) will be regenerated every time.

locking_timeout This is maximum time in seconds a cache lock is valid to avoid dead locks. The deafult value is 10 seconds.

merge_compiled_includes By setting `$merge_compiled_includes` to `TRUE` Smarty will merge the compiled template code of subtemplates into the compiled code of the main template. This increases rendering speed of templates using a many different sub-templates. Individual sub-templates can be merged by setting the inline option flag within the `{include}` tag. `$merge_compiled_includes` does not have to be enabled for the inline merge.

php_handling This tells Smarty how to handle PHP code embedded in the templates. There are four possible settings, the default being `Smarty::PHP_PASSTHRU`. Note that this does NOT affect php code within `{php}{/php}` tags in the template. Settings: `Smarty::PHP_PASSTHRU` - Smarty echos tags as-is; `Smarty::PHP_QUOTE` - Smarty quotes the tags as html entities; `Smarty::PHP_REMOVE` - Smarty removes the tags from the templates; `Smarty::PHP_ALLOW` - Smarty will execute the tags as PHP code.

plugins_dir This is the directory or directories where Smarty will look for the plugins that it needs. Default is `plugins/` under the `SMARTY_DIR`. If you supply a relative path, Smarty will first look under the `SMARTY_DIR`, then relative to the current working directory, then relative to the PHP include_path. If `$plugins_dir` is an array of directories, Smarty will search for your plugin in each plugin directory in the order they are given. **While using the SmartyBundle you may add plugins by setting services tagged with smarty.extension. See section Extensions for more information.**

smarty_debug_id The value of `$smarty_debug_id` defines the URL keyword to enable debugging at browser level. The default value is `SMARTY_DEBUG`.

template_dir This is the name of the default template directory. If you do not supply a resource type when including files, they will be found here. By default this is `%kernel.root_dir%/Resources/views`. `$template_dir` can also be an array of directory paths: Smarty will traverse the directories and stop on the first matching template found. **Note that the SmartyEngine included in this bundle already add the template directory of each registered Bundle**.

trusted_dir `$trusted_dir` is only for use when security is enabled. This is an array of all directories that are considered trusted. Trusted directories are where you keep php scripts that are executed directly from the templates with `{include_php}`.

use_include_path This tells smarty to respect the `include_path` within the `File Template Resource` handler and the plugin loader to resolve the directories known to $template_dir. The flag also makes the plugin loader check the `include_path` for `$plugins_dir`.

use_sub_dirs Smarty will create subdirectories under the compiled templates and cache directories if $use_sub_dirs is set to `TRUE`, default is `FALSE`. In an environment where there are potentially tens of thousands of files created, this may help the filesystem speed. On the other hand, some environments do not allow PHP processes to create directories, so this must be disabled which is the default. Sub directories are more efficient, so use them if you can. Theoretically you get much better perfomance on a filesystem with 10 directories each having 100 files, than with 1 directory having 1000 files. This was certainly the case with Solaris 7 (UFS)... with newer filesystems such as ext3 and especially reiserfs, the difference is almost nothing.

CHAPTER

EIGHT

AUTHORS

Vítor Brandão - noisebleed@noiselabs.org ~ twitter.com/noiselabs ~ blog.noiselabs.org

See also the list of contributors who participated in this project.

CHAPTER

NINE

SUBMITTING BUGS AND FEATURE REQUESTS

Bugs and feature requests are tracked on GitHub.

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*