# Documentation

## HTML_Template_Xipe v1.7

Creation date: 21 Jun 2002
Last change: 8 Feb 03
Author: Wolfram Kriesing
<wolfram@kriesing.de>

# 1 HTML_Template_Xipe Overview

This is a template engine which offers all the features you have when using PHP, without a big effort to learn a new template language.

Some small examples will demonstrate how the engine works and may be it gives you a short overview on how the concept works:

The template file (index.tpl)
*options: autoBraces=false*

```
<html>
    <title>{$title}</title>
    <body>
        Welcome {$username}!
        You can choose from:
        {foreach( $list as $aItem ):}
            <li>{$aItem}</li>
        {endforeach}
        <br />
        Please don't hesitate to choose the right thing :-)
    </body>
</html>
```

Writing templates this way is probably the most common use. The rules are simple, the annoying <?php ?> tags are replaced by { }. That's almost all there is to say. So why use it at all? Xipe provides an environment to handle caching and a very high degree of customization, either of the delimiters, the cache handling, multilingual features and much more!

The source code file (index.php):

```
<?php
    require_once 'HTML/Template/Xipe.php');
    $options =  array( 'templateDir'     =>      dirname(__FILE__),
                       'autoBraces'      =>      false);
    $tpl    =  new  HTML_Template_Xipe($options);
    $tpl->compile('index.tpl');
    include($tpl->getCompiledTemplate());
?>
```

The template file (index.tpl)
*options: autoBraces=true*

```
<html>
    <title>{$title}</title>
    <body>
        Welcome {$username}!
        You can choose from:
        {foreach( $list as $aItem )}
            <li>{$aItem}</li>
        <br />
        Please don't hesitate to choose the right thing :-)
    </body>
</html>
```

As you can see right away in the template file the blocks are built by proper indention, the foreach has no implicit end-tag!

The source code file (index.php):

```
<?php
   require_once 'HTML/Template/Xipe.php');
   $options =  array( 'templateDir'      =>      dirname(__FILE__));
   $tpl    =  new  HTML_Template_Xipe($options);
   $tpl->compile('index.tpl');
   include($tpl->getCompiledTemplate());
?>
```

## 1.1 Features

The template engine is a **compiling engine**, all templates are compiled into PHP-files. This will make the delivery of the files faster on the next request, since the template doesn't need to be compiled again. If the template changes it will be recompiled.

There is no new **template language** to learn (chapter 5), you can use standard php as you already know it. Why I did it this way, read the chapter before.

By default the template engine uses **indention** for building blocks (chapter 5.1.1). This feature was inspired by Python and by the need I felt to force myself to write proper HTML-code, using proper indentions, to make the code better readable.

**Every template is customizable** in multiple ways (chapter 8). You can configure each template or an entire directory to use different delimiters, caching parameters, etc. via either an XML-file or a XML-chunk which you simply write anywhere inside the tpl-code.

Using the **Cache** (chapter 9) the final file can also be cached (i.e. a resulting HTML-file). The caching options can be customized as needed. The cache can reduce the server load by very much, since the entire php-file doesn't need to be processed again, the resulting client-readable data are simply delivered right from the cache (the data are saved using php's output buffering).

The template engine is prepared to be used for **multi-language** applications too (chapter 10). If you i.e. use the PEAR::I18N (see http://pear.php.net) for translating the template, the compiled templates need to be saved under a different name for each language. The template engine is prepared for that too, it saves the compiled template including the language code if required (i.e. a compiled *index.tpl* which is saved for english gets the filename *index.tpl.en.php* ).

# 2 The history and the future

While searching THE template engine for my uses i came across many of them and I have worked with all the very known ones for a while until I realized that none of them offers exactly what I needed or that at least not one of them has all the features at once.

## 2.1 Testing existing template engines, Pro's and Con's

Smarty is caching the pages and it is the most powerful template engine, everyone says. So that was the way to go. But I got stuck quite soon, when trying to port a bigger application to use Smarty. There was no way using Smarty to refer to an array using a template variable as a key (I only needed something like this inside a template: *{$array[$aVar]}* ). I was asking on the mailing list and was told that this is not possible (yet).

IT[X] is in my eyes not really meant for high traffic sites, since it parses the template every time and the programming to control the template became to complicated for bigger sites. The 'setCurrentBlock', 'setVariable' and alikes were taking up almost as much code as the business logic itself.

What was bothering me about all the existing template engines was that you always had so much code inside the php-file which was needed to either assign the values and/or to build the control structures for showing the data. I only want the business logic inside the php-files no additional code which controls the template. For me the template was supposed to use and show the data generated by the business logic in whatever way needed.
Some of the aspects might not be very clean from a MVC-point of view perspective but it seems most practical in my eyes. So may be I just wanted something in between.

Initially it was not my intention to write a template engine, I wanted to use the best that exists, work peacefully and let others take care of maintaining the tool.
But for the reasons mentioned above, one night I came up with the idea to write the simplest template engine on earth. That's why the actual name was 'SimpleTemplate'.

## 2.2 The 'simplest' template engine on earth

This template engine shall do nothing more than provide all the power of PHP inside the template without writing the ugly *<?php* and *?>* tags. Actually i only wanted to replace those by some delimiter which is easier to write and looks nicer, like {% and %}. So the template engine should not do anything else but parsing a php-file, which has {% and %} instead of *<?php* and *?>* in it, and replace those {% and %}. That was the entire idea.
A template could be this:

```
<html>
```

```
    <title>{%echo $title%}</title>
    <body>
        Welcome {%echo $username%}!

        {%foreach( $list as $aItem ) {   %}
            <li>
                {%echo $aItem%}
            </li>
        {%  } %}
    </body>
</html>
```

This would provide all the power of php inside the template. To make it all work like a real template-concept you only need to separate the actual php-code from the template and that's a template concept. So it just needs a little self-discipline and the template engine works like all the others.

Since the template file is simply included (after compiling) at the end of the php-file you don't need no assign-methods. Any variable used in any code before is available in the template too.
So the php-file could be like this:

```
    // $title can be used in the template file without assigning or
    // anything
    $title = 'Template_Xipe - translating template engine';
    $username = 'Wolfram Kriesing';
    $list = array( 'easy' , 'compiling' , 'translating' , 'and so on' );

    $tpl->compile( <template file> );
    include( <compiled tempalte> );
```

So that the PHP-interpreter would finally get something like this:

```
<?php
    // $title can be used in the template file without assigning or
    // anything
    $title = 'Tempalte_Xipe - translating template engine';
    $username = 'Wolfram Kriesing';
    $list = array( 'easy' , 'compiling' , 'translating' , 'and so on' );

    $tpl->compile( <template file> );
    include( <compiled tempalte> );        // as attached below
?>
<html>
    <title><?php echo $title ?></title>
    <body>
        Welcome <?php echo $username ?> !

        <?php foreach( $list as $aItem ) {   ?>
            <li>
                <?php echo $aItem ?>
            </li>
        <?php } ?>
    </body>
</html>
```

This way I had all the problems solved the template engines I tested before gave me:

- referencing arrays using variables as keys, or any other wild referencing thing, is no problem

- no assign methods needed
- simple template logic in the template file, more than *foreach* and *if* wont be needed, but still everything is available (while debugging a *print_r* in the template is no problem either)
- no complex parsing and converting of variables and files is needed
- and the compiled template is cached so it only needs to be recompiled when it changes.

## 2.3 'Nice to haves' everyone needs

The start was made and as every programmer knows it, once you start programming you can always make the existing thing better and implement features which seem useful.

So I started with those which seemed most obvious to me.

### 2.3.1 Auto braces

As you can see in the code you still need to make the php-blocks, (i.e. for the *foreach*) using the brackets { and }.

```
{%foreach( $list as $aItem ) {  %}
    <li>
        {%echo $aItem%}
    </li>
{% } %}
```

But if you simply look at the code the indention would do the work too. So I recalled, that the programming language Python has this feature too, so it can't be so wrong (and I like it). Python builds the blocks (which are delimited using { and } in PHP) by the indention.
So I implemented the 'autoBrace' feature, which makes this code work (note the missing { and } for building the php blocks, the indention is used instead):

```
{%foreach( $list as $aItem )%}
    <li>
        {%echo $aItem%}
    </li>
```

A nice side effect is that the template code **has to be** nicely formatted.

### 2.3.2 Customizable delimiters and printing variables directly

Now that the blocks are built using the indention, the delimiters { and } are not reserved anymore, since they are not needed anymore. So I can use them as the delimiters.

Every template engine has a simple way of printing variables, so I implemented one too, so the *echo* wont be needed anymore. Now every delimiter followed by a $-sign is converted to be echoed, as you can see in the following code.

```
{foreach( $list as $aItem )}
    <li>
        {$aItem}
        </li>
```

The delimiter became customizable of course.

### 2.3.3 File handling

Meanwhile the entire file handling had to be implemented. I wanted to use the template engine in my existing projects. Therefore the compiled templates had to saved in unique places, the re-compiling had to be triggered properly and some other things such as the option *forceCompile* had to be implemented.

## 2.4 Translating, multilanguage applications

After developing a big site which had to be multilingual I was quite disappointed about the existing tools for developing multilingual web applications. We did it back then using the worst way possible, copying the entire application and translating it. Then I tried the gettext-way. I didnt want no {$TEXT123} which then is replaced by the text which is in some external data source. This makes the code very ugly in my eyes.

Then I thought of the easiest solution: simply translate the existing templates using regular expressions and apply a post filter, which translates the dynamic text using the same data source.

Since I am compiling the template anyway I can also look into the template and try to translate the text that seems translatable. For example:

```
<html>
   <body>
      Welcome {$username}!
      <br/>
      Have a nice day and enjoy our site!
...
```

In this template you can see, that for translating it you would only need to pick out the strings 'Welcome ... !' and 'Have a nice day and enjoy our site!'. So the translated and compiled template, i.e. into german will look like this:

```
<html>
   <body>
      Willkommen <?php echo $username ?>!
      <br/>
      Viel Spaß auf unserer Seite und einen schönen Tag noch!
...
```

This doesn't seem a big task, simply replace all english text by all the german text, or any other language. Now we only have to save every compiled template for each language in a different file, so that *index.tpl* will be compiled and saved as *index.tpl.en.php* and *index.tpl.de.php*. Now we have compiled the template-tags and converted them into proper PHP and additionally we have translated the text that we could translate. We don't need no ugly place holders or constants which represent our text, we can simply write the text inside the template and the translation mechanism translates all that you tell it to.

## 2.5 The future

one day $day wont need a delimiter around it, like velocity

# 3 What does it do

Tbd

## 3.1 How

The template engine processes all files using regular expressions and simple string replacing.

# 4 Installation

## 4.1 The template engine

Download the latest version of the template engine from http://sourceforge.net/projects/simpletpl.

Extract the package and move it in your include path. Your php-include path is the path that PHP uses to search for files, you can define it in your *php.ini* under the key *include_path*.

## 4.2 Optional packages

Optional packages are:

| | |
|---|---|
| for XMLConfig: | PEAR::Tree |
| for translating: | PEAR::Tree and PEAR::I18N |

If you are using the feature XML-Config you need to install the PEAR package Tree.

For translation as described in a later chapter you need to download the Tree package and the I18N package from the pear repository.

### 4.2.1 Installing using the PEAR-Installer

Install them using the pear-installer. It will download the packages from the pear-server and install them in your pear directory, which by default is also in your php-include path.

To check if you have the pear-installer on your system simply type *pear* on the command line, if the command can be found you have pear installed properly. Now you should be able to install the packages by typing

```
>pear install Tree
```

and

```
>pear install I18N
```

### 4.2.2 Installing without the PEAR-Installer

It is not suggested to install those packages without the pear-installer. You should only do this when you know exactly what you are doing.

Download the two packages from http://pear.php.net. Extract the files and copy the directories in your include path. You might need to remove the version numbers from the directory name.

# 5 Template languages

Since version 1.6 another kind of language constructs are included in HTML_Template_Xipe , besides the standard language, the so called SimpleTag language (chapter 5.2).

## 5.1 Standard language

Actually there is no new made-up template language to learn. The language you use to implement the logic inside the templates is standard PHP. Some optimizations for the use as a 'template language' do exist though.

All the power of PHP available inside a template may seem against the philosophy of templates. But my thought when doing it this way was to rather provide a powerful template language, which to use with great discipline is up to be determined by everyone itself, instead of having to implement every feature that might be requested one day.

If one says in a template language only *foreach* and *if* should be allowed, then this person has all the freedom to use only this subset. Furthermore one can also write a simple filter which prevents from the use of other features. The general intention was to provide all possible. Another advantage is that the template language doesn't need to be extended or changed once anything changes in PHP or some new PHP-functionality should become available, it simply is ready to use.

To get a quick overview here some examples:

```
<html>
   <body>
      {$text}             equivalent <?=$text ?>, prints the value of $text
                          because the $-sign follows right after the {

      { $text = 'Hello '}  equivalent <?php $text='Hello' ?>
                          notice the space between the { and the $-sign


      Auto braces

      {if( $x==1 )}
          x is 1
      {else}
          x is not 1 but
          x is {$x}

      {foreach( $allProducts as $aProduct )}
          Product name:   {$aProduct['name']}
          Product id:     {$aProduct['id']}
          Order number:   {$aProduct['orderNumber']}
      this string is already outside of the block and will only be show once.

   </body>
</html>
```

{$var=7}   prints '7'

{ $var=7}   only assigns 7 without printing, note the space in front!!!

### 5.1.1 Auto braces

By default the template engine builds the blocks (as needed in php) by indention.

```
{if( $x==1 )}
    x is 1
{else}
    x is not 1 but
    x is {$x}
```

or

```
{foreach( $allProducts as $aProduct )}
    Product name:  {$aProduct['name']}
    Product id:       {$aProduct['id']}
    Order number:  {$aProduct['orderNumber']}
this string is already outside of the block and will only be show once.
```

NOTE: Be sure to indent lines which start with PHP-code only when you want to build a block. And be sure to use spaces instead of TABS for the indention!

Here are some examples where auto braces has an effect and builds a block

```
{if( $thisIsTrue() )}
    then print this :-)

{if( sizeof($anArray) )}
    {foreach( $anArray as $aElement )}
        {$aElement}
        <br/>

{if( is_array($anArray) && sizeof($anArray) )}
    well it is an array with content
    {if( sizeof($anArray) > 10)}
        there are more than 10 elements inside
        {{if( sizeof($anArray) > 20)}
            there are more than 20 elements inside
            OK here they are :-) <br/>
            {foreach( $anArray as $aElement )}
                {$aElement}
                <br>


ATTENTION: be careful with the following. Blocks will be built too, but I am
sure they are not meant to be built, and the indention makes no sense
anyway (in my eyes).

{print_r( $someVar )}
    <br>

{ $aVar = 'value'}
    { $anotherVar}
```

and those are examples where auto braces doesn't do anything

```
No blocks is built because the { is not at the beginning of the line
<form action="{$_SERVER['PHP_SELF']}">
    <input>
</form>

No block is built, because the {$formStart} is meant to print out a value
```

```
{$formStart}
    <input>
{$formEnd}
```

## 5.2 SimpleTag language

Since version  1.6

This is a set of filters which enable language constructs that can be used as an alternative to the standard template language described before. Please note that you can only use either one.

This language enables designers to finally use the template engine too. A short example demonstrates how this language works.

```
<!-- configure the template engine -->
<HTML_Template_Xipe>
    <options>
        <autoBraces value="false"/>
    </options>
    <preFilter>
        <register class="HTML_Template_Xipe_Filter_SimpleTag"/>
    </preFilter>
</HTML_Template_Xipe>

<!-- the actual template content starts here-->

#if( sizeof($task) )
    #foreach($task as $aTask)
        {$aTask}
        <br>
    #end
#end

#foreach(u::test() as $aTask){$aTask}<br>#end
```

The last line demonstrates one of the main differences to the standard language. The 'autoBraces' is and has to be turned of for this language (SimpleTag language).

This language is intended to provide control structures without depending on the format of the source code. So it can be used with WYSIWYG-tools such as dreamweaver etc.

As you can see in the source code the control structures start with a '#' and have to be terminated by a '#end'. Variables are still surrounded by the delimiters that you can configure as you want to.

Besides the mentioned differences, this language has all the features the standard language offers too. You can use pure PHP inside your templates just as you are used to.

As you can see in the example above, you have to configure the template engine to use this language. This does not have to be done for each template, you can simply put the XML-Part inside a *config.xml* file, which you place in the template-Root and the all templates which are in this path will use the settings from this *config.xml*, as long as they are not overwritten, see chapter 8 for more info on how to configure  the template engine via XML.

This is an example of a config.xml for using the SimpleTag language.

```
<!-- configure the template engine -->
<HTML_Template_Xipe>
    <options>
        <autoBraces value="false"/>
    </options>
    <preFilter>
        <register class="HTML_Template_Xipe_Filter_SimpleTag"/>
    </preFilter>
</HTML_Template_Xipe>
```

Be sure to have the option 'XML-Config' turned on, otherwise this *config.xml* file won't have no effect.

# 6 Setup

This chapter explains how to setup the template engine and how to get started using it.

TO BE UPDATED, since Xipe is in PEAR now!!!!

FOR NOW install it via the PEAR-installer:

use:     pear install HTML_Template_Xipe

## 6.1 Include path

Place the complete directory *SimpleTemplate* and all its subfolders in a directory which is in your PHP-include_path, you can define this in the file *php.ini*. If you dont have the permission to change the php.ini or you dont like to put this in the include path you can use

```
ini_set( 'include_path' , '.:/path/to/directory' )
```

if you also want to preserve the current include_path and only want to add the SimpleTemplate to the include path use

```
ini_set( 'include_path' , ini_get('include_path').':/path/to/directory' ).
```

Assuming you have copied SimpleTemplate in the directory '/usr/local/share/SimpleTemplate' your ini_set command would look like this:

```
ini_set( 'include_path' , '.:/usr/local/share' )
```

NOTE: On windows systems the path seperator is not a colon ':' as used here but a ';'.

## 6.2 Usage

A standard way to get the template engine to work is the following

```
require_once('HTML/Template/Xipe.php');
$options = array(  'templateDir'=> dirname(__FILE__),
                   'compileDir'  => 'tmp'          );
$tpl = new HTML_Template_Xipe($options);
```

The two options that are defined here are those which need to be given at least. The exact meaning is described in the next chapter.

### 6.2.1 The options

This chapter describes all the options you can set for the template engine. There are multiple ways of setting those options. Those will be described at the end of this chapter. The first and mostly used way is probably the one as shown in the example above, by passing those options to the constructor.

NOTE: Every option described in the following can also be passed to the constructor in the options array as shown above.

For demonstrating ways to set options we assume, that the instance of the template engine is named '$tpl'.

#### 6.2.1.1 *templateDir*

| | |
|---|---|
| Type | string |
| Default value | '' |

The first one *templateDir* is the one which defines the root directory in which your templates are stored.

The *tempalteDir* can not be changed using the XML-Config.

#### 6.2.1.2 *compileDir*

| | |
|---|---|
| Type | string |
| Default value | 'tmp' |

The *compileDir* is the directory where all the compiled templates will be stored.

NOTE: Be sure to give write rights (in the *compileDir*) to the user your webserver is running as, so it can write the compiled files in this directory.

The *compileDir* can not be changed using the XML-Config.

#### 6.2.1.3 *delimiter*

| | |
|---|---|
| Type | array |
| Default value | array('{','}') |

This option is an array, and it defines the delimiters used inside your tempaltes. By default this option has the following values and can be set

```
$tpl->setOption( 'delimiter' , array('{','}') );
```

You can set any delimiter you like, but you should be careful choosing the delimiters, so they dont make life hard when writing the source code of the tempaltes.

The delimiters can be any kind of string. But be careful not to use any that are often used inside PHP or HTML, because then you would have to escape them everytime you need for something else then a delimiter.

Setting the delimiters using XML

```
<HTML_Template_Xipe>
   <options>
      <delimiter begin="{%{" end="}%}"/>
   </options>
</HTML_Template_Xipe>
```

The XML chunk above sets the opening delimiters to '{%{' and the closing delimiters to '}%}'.

#### 6.2.1.4 *filterLevel*

| | |
|---|---|
| Type | integer |
| Default value | 10 |
| Since version | 1.5. |

This option defines which filters are used by default. It was added thanks to the hint from Alan Knowles. The *filterLevel* relieves you from the work of instanciating objects of each filter class and assigning the methods manually.

Still you can always add other filters from those shipped with the HTML_Template_Xipe classes or you can program your own filters. This is described in a later chapter.

For now there are only some filterLevels defined, those are

| ***filterLevel*** | ***Comment*** |
|---|---|
| 0 | Applys no filters at all. The code is being compiled and not changed in any way. TagLib-tags are also not available, since the TagLib also works as a filter. |
| 1-7 | Not in use yet |
| 8 | Like level 9 only that all comments also stay inside the code. Actually no code optimization is done. You can use this mode to look at the compiled code to double check or investigate how the engine genreates the code and what it changes. |
| 9 | Like level 10 only, that all post filters, that optimize and modify the compiled template do not apply. This is very useful when debugging and when it is necessary to look at the code the tempalte engine has compiled. |
| 10 | DEFAULT. Applies all the filters that are available and make most sence to be applied, for a complete list see text below. |

For level 10 the following filters are applied

Basic-filters

removeHtmlComments
removeCStyleComments
addIfBeforeForeach
removeEmptyLines
trimLines
optimizeHtmlCode

TagLib

includeFile
block
macro
trimByWords
trim
repeat
applyHtmlEntites

For level 9 all filters from level 10 but the following apply

removeEmptyLines

trimLines
optimizeHtmlCode

For level 8 all filters from level 9 but the following apply

removeHtmlComments
removeCStyleComments

This option is not yet configurable via XML.

#### 6.2.1.5 *forceCompile*

| | |
|---|---|
| Type | boolean |
| Default value | false |

This option forces that the templates are compiled on every run. It is suggested to use this feature only while debugging or during development. On a live server you should set it to false, since that increases speed.

```
$tpl->setOption( 'forceCompile' , true );
```

This option can not be set via XML.

#### 6.2.1.6 *logLevel*

| | |
|---|---|
| Type | integer |
| Default value | 1 |

Using this option you can set the level of logging, this means how much or if at all is written in the log files. A log file is created for each template file individually in the compileDir.

In the highest logLevel 3 those can be used to see which filters were applied, how long it took to proceed each filter and the size of the file after processing the filter.

The logLevels mean

| ***logLevel*** | ***Comment*** |
|---|---|
| 0 | Logs nothing and creates no log files at all, use this in a live environment. |
| 1 | Only logs new compiles. |
| 2 | Logs everything even only deliveries of a template file. |

#### 6.2.1.7 *enable-XMLConfig*

| | |
|---|---|
| Type | boolean |
| Default value | false |
| Since version | 1.5 |

Setting this option to true gives you the opportunity to configure each template or all templates in a directory and its subdirectories.

See chapter ... on how to use the xml tags to configure your template(s).

You can set the option using

```
$tpl->setOption( 'enable-XMLConfig' , true );
```

#### 6.2.1.8 *enable-Cache*

| | |
|---|---|
| Type | boolean |
| Default value | false |
| Since version | 1.5 |

This option enables the caching of templates.

See chapter ... on how to use caching.

To enable caching you can use

```
$tpl->setOption( 'enable-Cache' , true );
```

#### 6.2.1.9 *autoBraces*

| | |
|---|---|
| Type | boolean |
| Default value | true |

This option takes care of creating the blocks using the indention in the template files. See also chapter .... for more infos about *autoBraces*.

```
$tpl->setOption( 'autoBraces' , false );
```

You can also set this option using the XMLConfig inside your template, be sure to have 'enable-XMLConfig' set to true for this to take effect.

```
<HTML_Template_Xipe>
    <options>
        <autoBraces value="false"/>
    </options>
</HTML_Template_Xipe>
```

If you turn *autoBraces* off you should be careful with using the curly braces ( '{' and '}' ) as delimiters, since they are used by PHP to build blocks.

Turning it off might make sense if you write filters which handle templates like the following (Velocity-like http://jakarta.apache.org/velocity )

```
#foreach ( $aValue in $someArray )
    #if  ( $aValue != '' )
        This is a value: $aValue
    #end
#end
```

or if you know exactly what you are doing.

If *autoBraces* is off and the delimiters are i.e. set to '[[' and ']]' then your template might look like this

```
[[ foreach($someArray as $aValue) ]]
[[ { ]]
    [[ if ( $aValue != '' ) {  ]]
```

```
        This is a value: [[ $aValue  ]]
    [[ } ]]
[[ } ]]
```

Note that this code snippet contains the block building curly braces. With the *autoBraces* turned on you wouldnt have to care about that.

Be sure to think about all the side effects this might have and take care of handling them properly.

#### 6.2.1.10 *makePhpTags*

| | |
|---|---|
| Type | boolean |
| Default value | true |
| Since version | 1.5 |

This option takes care of converting the delimiters to proper php tags. Turning this option off (setting it to false) makes only sense if you want to use the template engine for a special reason, or if you write your own filter(s) which do(es) the parsing and take(s) care of all the conversions.

```
$tpl->setOption( 'makePhpTags' , false );
```

#### 6.2.1.11 *xmlConfigFile*

| | |
|---|---|
| Type | string |
| Default value | 'config.xml' |

The string given to this option is the filename of the XML-config file that will be used to configure the templates.

#### 6.2.1.12 *locale*

| | |
|---|---|
| Type | string |
| Default value | 'en' |

When you are using the translation feature this string will be added to the compiled template file. If you are not using the translation feature you can simply set this option to an empty string ('').

If this value has the default value 'en' a template with the name *index.tpl* will be saved as *index.en.tpl.php* after compilation.

#### 6.2.1.13 *cache*

| | |
|---|---|
| Type | array |
| Default value | |

```
        array( 'time'     =>    false,
               'depends'=>    0).
```

| | |
|---|---|
| Since version | 1.5 |

This value contains multiple values, which are saved in an array. When instantiating an instance of the template engine and passing this option, you can determine the default behavior of the caching mechanism. If you don't configure templates to be cached with different settings those values will be used.

Setting this value only has an effect when 'enable-Cache' is turned on too.

set this to the number of seconds for which the final (html-)file shall be cached

// false means the file is not cached at all

// if you set it to 0 it is cached forever

,'depends' => 0

// what does it depend on if the cache file can be reused

// i.e. could be $_REQUEST $_COOKIES

```
<HTML_Template_Xipe>
   <options>
      <cache>
         <time value="10" />
         <depends value="$_REQUEST['listStart']"/>
      </cache>
   </options>
</HTML_Template_Xipe>
```

OR

```
<HTML_Template_Xipe>
   <options>
      <cache dontCache="true" />
   </options>
</HTML_Template_Xipe>
```

#### *6.2.1.14 verbose*

| | |
|---|---|
| Type | boolean |
| Default value | true |

The default value is true to print out error messages right away, to make setting up the template engine easier. Those error messages actually only appear for problems that need to be solved before getting the template engine running properly, such as:

- a missing compileDir
- no write right in the compileDir, etc.

On a production system you should turn this option off!

#### 6.2.1.15 *logFileExtension*

| | |
|---|---|
| Type | string |
| Default value | 'log' |

This is the extension that will be added to the log files. Note that log files will only be written when the option 'logLevel' is set to do so.

#### 6.2.1.16 *cacheFileExtension*

| | |
|---|---|
| Type | string |
| Default value | 'html' |

This is the file extension for the cached files. Files will only be cached when the 'enable-Cache' option is true.

#### 6.2.1.17 *debug*

| | |
|---|---|
| Type | integer |
| Default value | 0 |

Set the debug level here.

### 6.2.2 Setting options

You can set options in different ways, which will be described in the following.

#### 6.2.2.1 *Passing to the constructor*

The most common way, and the best practice is to set the options passing them to the constructor, as shown in the code example at the beginning of this document and as done in most of the examples.

#### 6.2.2.2 *Call setOption*

```
$tpl->setOption('templateDir','/path/to/the/templateDir');
```

#### 6.2.2.3 *XML-config*

Currently you can not set every option using the XML-config.

```
<HTML_Template_Xipe>
   <options>
      <delimiter begin="{%{" end="}%}"/>
      <cache>
         <time value="10" />
         <depends value="$_REQUEST['listStart']"/>
      </cache>
   </options>
</HTML_Template_Xipe>
```

### 6.2.3 Getting options

Get a option, which is a scalar value.

```
$tpl->getOption('templateDir');
```

Getting an option which is an array.

```
$cacheTime =  $tpl->getOption('cache','time');  // get the caching time
```

This retreives the option 'time' which is saved in the option array 'cache'. See chapter .... for an example.

```
$openDel    =  $this->getOption( 'delimiter' , 0 )        //    the    opening
delimiter
$closeDel   =  $this->getOption( 'delimiter' , 1 )        //    the    closing
delimiter
```

# 7 Filter

Pre and postfilters ...

## 7.1 Basic Filters

Those are filters which modifiy the template before saving it as the compiled template.

## 7.2 TagLib Filters

Named after the JSP TagLibs

All TagLib-filters start with the delimiter followed by a '%' and they also end with a '%' sign followed by the end delimiter. Assuming your delimiters are '{' and '}' a TagLib-tag would look like this

```
{%methodName parameters %}.
```

TagLibs are different from the basic filters because they have additional functionality implemented, i.e. trimming strings. TagLib-tags are mostly configurable and useable in different ways for different tasks.

### 7.2.1 Trim and trimByWords

This trims the variable given after the number of characters you specified.

Examples for using it:

```
{%trim $text 10 “...”%}
```

This trims *`$text`* after 10 characters and if the *`$text`* was longer than 10 characters it adds “...” at the end.

### 7.2.2 Repeat

The repeat-tag is just an easy way for replacing a for-loop which only has to go through a block of code a couple of times. The syntax is quite simple:

```
{%repeat $count [$varibaleName]%}
```

It repeats the block below the repeat-tag *`$count`* times and uses the varibale *`$variableName`* to let you access the current counter value. Where it doesn't matter if you write any more words inbetween, i.e. You could write *`{%repeat 7 times%}`* and it would repeat the block below seven times, this is allowed to make the entire code more readable. Also you can write *`{%repeat 7 times with $x%}`* where *`$x`* will be the counter varibale, which you can use to access the current value of the loop.

Examples are as follows:

```
{%repeat 3%}
   repeat me
```

```
{%repeat 3 times%}
    repeat me
```

each of the above repeats the text 'repeat me' 3 times

```
{%repeat $count times%}
    repeat me

{%repeat $obj->methodCall() %}
    repeat me

{%repeat sizeof($x)%}
    repeat me

{%repeat $count times $counter%}
    repeat me

{%repeat $count times using $counter%}
    repeat me
    <br/>
```

each of the above repeats the block below the repeat-tag, as many times as the variable behind *repeat* tells.

### 7.2.3 Include

The include-tag can be used to include other templates, which will be parsed too. This is very useful for including templates or especially macros. The actual reason why I implemented this was that I wanted to have macros which do common functionality such as creating some form tags etc. The syntax is as follows:

```
{%include /path/to/file.tpl%}
```

The file is searched for in the 'templateDir' as defined in your options and in the php-include path, set either in the php.ini or via ini_set, etc.

### 7.2.4 Macro

## 7.3 Internal Filters

Those are only used by the template engine itself. Normally it doesn't make much sense using them directly, but who knows what people come up with :-).

## 7.4 Modifier Filters

Check out imgSrc-Filter this really saves a lot of time :-)!!!!

## 7.5 Applying filters

### 7.5.1 Creating custom filters

Extend the class HTML_Template_Xipe_Options so you can also access all the options set for the template engine. If you are writing prefilters you will surely need to get the currently used delimiter, which you can do by calling the method getOption in the following way

```
$openDel = $this->getOption( 'delimiter' , 0 )    // the opening delimiter
$closeDel = $this->getOption( 'delimiter' , 1 )    // the closing delimiter
```

# 8 XML config

Override from template root to the tempalte file

## 8.1 Available tags

For now the following tags are available, each of them is optional.

```
<HTML_Template_Xipe>
    <options>
        <delimiter begin="{%{" end="}%}"/>
        <autoBraces value="false"/>
        <locale value="en"/>
        <cache>
            <time value="10" />
            <depends value="$_REQUEST['listStart']"/>
        </cache>
    </options>
</HTML_Template_Xipe>
```

Since version 1.5.2 you can also register filters using the xml-config. This is just a partly implementation now, so it will only work as explained here. Here is an example:

```
<HTML_Template_Xipe>
    <preFilter>
        <register class="SimpleTemplate_Filter_SimpleTag"/>
    </preFilter>
</HTML_Template_Xipe>
```

If 'class' is given it's assumed, that the class name represents also the path name, so `HTML_Template_Xipe_Filter_SimpleTag` means the file is in *HTML/Template/Xipe/Filter/SimpleTag.php* which means (mostly) the file has to be in the include path!
If the attribute 'classFile' is given it will be used for including the proper file instead of 'assuming' the filename. So you could also use the attribute `classFile="HTML/Template/Xipe/Filter/SimpleTag.php"`.

## 8.2 Configuring a single template

## 8.3 Configuring multiple templates

Either those in a directory and all template in the subdirectories

# 9 Caching

Caching means that the resulting pages which is delivered to the client will be cached. In case of websites those can be HTML files. You can configure for how long pages shall be cached and what the cached file depends on.

If caching is enabled it can reduce the server load by much. If you have a very frequently visited site, which always retrieves a lot of data from the DB to create the page, caching this page will reduce the load of PHP processing and the DB-load.

To enable caching set the option 'enable-Cache' to true. This can only be done by passing this option to the constructor. The xml configuration has to be enabled too, but you don't need to do that explicitly because the 'enable-Cache' does that already.

Add this parameter to the options array:

```
      'enable-Cache'  =>    true
```

And use the XML-Config (for more info see chapter 8) to determine the caching parameters to configure each template or multiple templates.

```
<HTML_Template_Xipe>
   <options>
      <cache>
         <time value="1" unit="day" />
         <depends value="$_REQUEST['listStart']  $anotherVar  $anyVar"/>
      </cache>
   </options>
</HTML_Template_Xipe>
```

```
<HTML_Template_Xipe>
   <options>
      <cache dontCache="true" />
   </options>
</HTML_Template_Xipe>
```

## 9.1 Setting cache parameters

You can set cache parameters in different ways, the most common one is as shown above, using the XML config.

Another way is passing the caching options to the constructor, as shown in the chapter on how to set the *cache* option.

## 9.2 Caching time

The caching time is the time when the cached file expires.

The default unit is seconds, but you can also use any other unit described here:

```
         <time value="1" unit="minute" />
```

```
        <time value="3" unit="hours" />
```

```
        <time value="2" unit="weeks" />
```

```
        <time value="1" unit="day" />
```

### 9.2.1 Traps :-)

Generating cache-pages (of one tpl) at different times so they also expire differently ... !!! i have to implement cache groups for that!

## 9.3 Dependencies

Dependencies are useful when you have a page which depends on the result of different variables or parameters. For example: a product catalog which shows the catalogs of different categories or depending on the current category to show the page always looks different. But all this can easily be achieved writing only one PHP page. So you need a mechanism to tell the cache that it has to cache the product page for category X in a different file then the page for category Y. This can be achieved by configuring the dependencies of a page. In the case of the example just mentioned it would simply be the category that the page depends on.

Configuring the page to depend on the category could be achived like this:
assuming the category is saved in the PHP-variable $category

```
<HTML_Template_Xipe>
    <options>
        <cache>
            <depends value="$category"/>
        </cache>
    </options>
</HTML_Template_Xipe>
```

assuming that the category is saved in the product array, the line has to be:

```
            <depends value="$product['category']"/>
```

A page might not only depend on one variable. You can also set a number of variables a page's caching shall depend on, simply seperate them by spaces

```
            <depends value="$category $company $product['id']"/>
```

or define multiple depend tags

!!!!! NOT yet implemented, at least not even tested if it might already work :-)

```
        <depends value="$category"/>
        <depends value="$company"/>
        <depends value="$product['id']"/>
```

Warning:

It is not suggested to let a page depend on data which can be passed to the page from the outside. Those kind of data mostly are the request data, such as GET, POST or COOKIE data. If you let the page depend on any of this data it could be very easy for an attacker to run a DOS (Denial of Service) attack, because the template engine has to generate a new page for all the different request parameters.

For example: http://home.com/index.php?depends=anything. The string 'anything' can be set to any value.

There are plans to prevent this from happening, by adding a maxCacheSize option to the template engine. If this size is reached the template engine will automatically turn the caching off and not produce any more cache files.

# 10 Translating

Tbd.