

Fluid: Templating Made Easy

New Template Engine for FLOW3 and TYPO3 4.x

Developers for TYPO3 usually rely on the popular marker based version which has been an integral part of TYPO3 for a long time when dealing with templates. Others have already worked with Smarty, PHPTAL or other template engines and now might wonder why there is need for a further template engine. An example about the output of a list of blogposts is meant to show, where the problems could be found: ↘ *Sebastian*

Kurfürst

Classical Templating

```
// Im Template

###CONTENTS###
<ul>
###SUBELEMENT###
<li>###TITLE###</li>
###SUBELEMENT###
</ul>
###CONTENTS###

// in PHP

$subEl = getSubpart("SUBELEMENT");
$out = ""; foreach ($recordList as $record) {
    $out .= substituteMarker($subEl, 'TITLE', $record['title']);
}
return substituteSubpart($template, 'SUBELEMENT', $out);
```

Listing 1

As one can see, there is a lot of code needed for a simple task such as outputting a list - on top of this there is no clear cut between the presentation logic and the application logic.

After it had become clear that TYPO3 would need a new templating solution for the future, the core-development team investigated many different template engines but all of them had some mutual disadvantages:

- ⊕ not completely written object orientated (PHP 4)
- ⊕ complicated and illogical syntax
- ⊕ templates that do not allow PHP
- ⊕ inadequate and blotted expandability

This is where Fluid [1] comes into play: The template engine has a new, intuitive syntax at its command which is extremely flexible and expandable while being easily learnable at the same time. The following example about a blog will show how efficiently developers can work with Fluid.

Traversing Objects Made Easy

Variable can be assigned within the controller with the help of „`$this->view->assign('blogTitle', 'Mein erster Blog');`“. In order to output the content of a variable in the template, the `{...}`-syntax is available: `{blogTitle}`.

Simple Output of Variables

```
// Im Controller:
$this->view->assign('blogTitle', 'Mein erster Blog');
```

```
// Im Template:
<h1>{blogTitle}</h1>
```

Listing 2

On top of that it is possible to “muddle through objects” by using a dot: “`{blog.title}`” leads to an invocation of “`$blog->getTitle()`”, “`{post.author.name}`” analogue to “`$post->getAuthor()->getName()`”.

Traversing Objects

```
// Im Controller:
$this->view->assign('blog', $blog);
// Im Template:
<h1>{blog.title}</h1>
```

Listing 3

Using Standard ViewHelper

Now we want to expand the example from above and output a list of blog postings. To achieve this, one can use so called “ViewHelper” in Fluid. They are invoked with the help of special tags within the template. The following listing outputs all articles of a blog:

Invoking ViewHelper

```
<f:for each="{blog.posts}" as="post">
  <h2>{post.title}</h2>
  <p>{post.teaser}
  <f:actionlink action="show" arguments="{post : post.uid}">mehr...</f:actionlink>
</p>
</f:for>
```

Listing 4

The special tag “`<f:for>`” is such a ViewHelper. With its help it is possible to iterate via arrays in order to output the individual elements. In the example above there is another ViewHelper, namely “`<f:actionlink>`” which is used for outputting a link to the detail view.

Fluid includes ViewHelpers which are needed regularly such as loops, links, forms or localization [2]. Additionally every extension can define and use its own ViewHelper.

Individual ViewHelper

Now we want to expand the blog example even further by implementing a ViewHelper for the online service “Gravatar”. Gravatar assigns an avatar image to an e-mail which is then displayed with each blog comment, where the e-mail address is provided. Hence the ViewHelper needs to accept an e-mail address and then should

generate the following schematic output: “”.

In order to be able to use a ViewHelper, it first needs to be imported before it actually is available in the template. Importing is undergone by using “{namespace blog=Tx_Blog_ViewHelpers}”. Now it is possible to invoke ViewHelper below “Tx_Blog_ViewHelpers” by using <blog:....>-tags within the template. (\$email)" />”.

The Gravatar ViewHelper is meant to be accessed in the template via “<blog:gravatar>”. This calls for an implementation within the class “Tx_Fluid_ViewHelpers_GravatarViewHelper”. The name of the class is composed of the namespace, the name of the tag (without the namespace prefix) and the ending “ViewHelper”.

Use of Gravatar ViewHelpers

```
{namespace blog=Tx_Blog_ViewHelpers}
<blog:gravatar email="{post.author.email}" />
```

Listing 5

Implementing Gravatar ViewHelper

Every ViewHelper needs to inherit from “Tx_Fluid_Core_AbstractViewHelper”. Additionally it needs to have a “render” method and its return value needs to be added to the output.

Hello World as ViewHelper

```
class Tx_Blog_ViewHelpers_GravatarViewHelper extends
Tx_Fluid_Core_AbstractViewHelper {
    public function render() {
        return 'Hello World';
    }
}
```

Listing 6

Now “Hello World” is output at the position in the template where “<blog:gravatar />” can be found. Since we need the e-mail address, we need to access the tag arguments.

Arguments Need to Be Registered

For security reasons all arguments need to be registered with name and data type - the framework then takes care of checking the arguments. Registering the arguments is simple:

Registering Arguments

```
/**
 * @param string $email Email to look for
 */
public function render($email) {
    ...
}
```

Listing 7

As you can see, all method arguments are registered automatically as tag attributes and are available within the render()-method. The block with comments above the method name is especially important because the type of the arguments (in this case “string”) is taken from there. The actual implementation of the render method is real easy afterwards:

Implementing the Render Method

```
class Tx_Blog_ViewHelpers_GravatarViewHelper extends
Tx_Fluid_Core_AbstractViewHelper {
    /**
```

```
* @param string $email Email to look for
 */
public function render($email) {
    return 'http://www.gravatar.com/avatar/' . md5(strtolower($email)) . '.jpg';
}
}
```

Listing 8

It is possible as well to define arguments as optional by the way: Just input standard values as you are used to from PHP:

Optional Arguments

```
// Kommentarblock nicht vergessen!
public function render ($email, $imageSize = 'large') {...}
```

Listing 9

Advanced Validation

By declaring the data type in “@param” one starts a simple validation. Unfortunately this is not enough in this case. The user can declare any string, but we only want to accept an e-mail address. Fluid offers enhanced validators for cases such as this one. They can be used via the @validate tag.

Enhanced Validation

```
/**
 * @param string $email Email of user
 * @param string $imageSize Size of the image.
 * @validate $email EmailAddressValidator
 */
public function render($email, $imageSize = 'large') {...}
```

Listing 10

Conclusion

Being an integral part of TYPO3 4.3 and FLOW3, Fluid allows flexible and nevertheless easy templating. At the same time it offers access to the concepts behind FLOW3 and TYPO3 5.0. ✖

Links and Literature [↗ Softlink 2461](#)

- [1] Forge-Projekt Fluid: <http://forge.typo3.org/projects/show/517>
- [2] Übersicht der Fluid-ViewHelper: http://forge.typo3.org/wiki/typo3v4-mvc/Fluid_v4_View_Helpers

The Author



Sebastian Kurfürst has been a TYPO3 core developer since 2005 and is especially interested in the interplay of TYPO3 v4 and v5. He studies computer science in Dresden and is a co-founder of sandstorm media (www.sandstorm-media.de).