

# ▼ Fluid Templating System

---

**Abstract** Fluid is a Templating System which is used by default for FLOW3-based applications.

**Author** Sebastian Kurfürst [sebastian@typo3.org](mailto:sebastian@typo3.org)

**Copyright** 2008 2009 Sebastian Kurfürst

## ▼ Chapter 1: Introduction

---

Fluid is a next-generation templating engine which was engineered with several goals in mind:

- Simplicity
- Flexibility
- Extensibility
- Ease of use

This templating engine should not be bloated, instead, we try to do it "The Zen Way" - you do not need to learn too many things, thus you can concentrate on getting your things done, while the template engine handles everything you do not want to care about.

### ▼ 1 What does it do?

In many MVC systems, the view currently does not have a lot of functionality. The standard view usually provides a `render` method, and nothing more. That makes it cumbersome to write powerful views, as most designers will not write PHP code.

That is where the Template Engine comes into play: It "lives" inside the View, and is controlled by a special `TemplateView` which instantiates the Template Parser, resolves the template HTML file, and renders the template afterwards.

#### ▼ 1.1 Example

Below, you'll find a snippet of a real-world template displaying a list of blog postings. Use it to check whether you find the template language intuitive (we hope you will ;-)

##### ▼ *Basic Fluid Example*

```
{namespace f=F3\Fuild\ViewHelpers}❶
<html>
<head><title>Blog</title></head>
<body>
<h1>Blog Postings</h1>
<f:for each="{postings}" as="posting">❷
  <h2>{posting.title}</h2>
  <div class="author">{posting.author.name} {posting.author.email}</div>❸
  <p><f:link.action action="details" arguments="{id : posting.id}">{posting.teaser}</f:
</f:for>
</body>
</html>
```

- ❶ The *Namespace Import* makes the `F3\Fluid\ViewHelper` namespace available under the shorthand `f`. This is important for View Helpers, like the `<f:link.action />` tag.
- ❷ This essentially corresponds to `foreach($postings as $posting)` in PHP.
- ❸ With the dot-notation (`{posting.title}`, or `{posting.author.name}`), you can traverse objects. In the latter example, the system calls `$posting->getAuthor()->getName()`.
- ❹ The `<f:link.action>...</f:link.action>` tag is a so-called *ViewHelper*. It calls arbitrary PHP code, and in this case renders a link to the "details"-Action.

There is a lot more to show, including:

- Layouts
- Custom View Helpers
- Boolean expression syntax

We invite you to explore Fluid some more, and please do not hesitate to give feedback!

## ▼ Chapter 2: User Manual

---

This chapter describes all things the users of the templating system needs to know. After you've read the introduction, you can dive into the concepts of Fluid which are relevant to you.

The chapter starts with an overview of basic concepts, continues with using layouts, and finishes with how to write your own view helpers.

### ▼ 1 Basic concepts

This section describes all basic concepts available.

This includes:

- Variables / Object Accessors
- View Helpers
- Arrays

#### ▼ 1.1 Variables and Object Accessors

A templating system would be quite pointless if it was not possible to display some external data in the templates. That's what variables are for:

Suppose you want to output the title of your blog, you could write the following snippet into your controller:

```
$this->view->assign('blogTitle', $blog->getTitle());
```

Then, you could output the blog title in your template with the following snippet:

```
<h1>This blog is called {blogTitle}</h1>
```

Now, you might want to extend the output by the blog author as well. To do this, you could repeat the above steps, but that would be quite inconvenient and hard to read.

**Footnote** Besides, the semantics between the controller and the view should be the following: The controller says to the view "Please render the blog object I give to you", and not "Please render the Blog title, and the blog posting 1, ...". That's why passing objects to the view is highly encouraged.

That's why the template language has a special syntax for object access, demonstrated below. A nicer way of expressing the above is the following:

```
This should go into the controller:↵
$this->view->assign('blog', $blog);↵
This should go into the template:
<h1>This blog is called {blog.title}, written by {blog.author}</h1>
```

Instead of passing strings to the template, we are passing whole objects around now - which is much nicer to use both from the controller and the view side. To access certain properties of these objects, you can use *Object Accessors*. By writing `{blog.title}`, the template engine will call a `getTitle()` method on the `blog` object, if it exists. Besides, you can use that syntax to traverse associative arrays and public properties.

**Tip** Deep nesting is supported: If you want to output the email address of the blog author, then you can use `{blog.author.email}`, which internally calls `$blog->getAuthor()->getEmail()`.

## ▼ 1.2 View Helpers

All output logic is placed in *View Helpers*.

The view helpers are invoked by using XML tags in the template, and are implemented as PHP classes (more on that later).

This concept is best understood with an example:

### ▼ Tags and Namespace declarations

```
<namespace f=F3\Fuild\ViewHelpers>❶
<f:link.action controller="Administration">Administration</f:link.action>❷
```

- ❶ **Namespace Declaration:** You import the PHP Namespace `F3\Fuild\ViewHelpers` under the prefix `f`.  
Hint: you can leave out this namespace import because it is imported by default.

**Footnote** This is like an XML namespace import.

- ❷ **Calling the View Helper:** The `<f:link.action...> ... </f:link.action>` tag renders a link.

Now, the main difference between Fluid and other templating engines is how the view helpers are implemented: *For each view helper, there exists a corresponding PHP class*. Let's see how this works for the example above:

The `<f3:link.action />` tag is implemented in the class `\F3\Fuild\ViewHelpers\Link\ActionViewHelper`.

The class name of such a view helper is constructed for a given tag as follows:

- The first part of the class name is the namespace which was imported (the namespace prefix `f` was expanded to its full namespace `F3\Fuild\ViewHelpers`)
- The unqualified name of the tag, without the prefix, is capitalized (`Link`), and the postfix `ViewHelper` is appended.

The tag and view helper concept is *the core concept* of Fluid. *All output logic is implemented through such ViewHelpers / Tags!* Things like `if/else`, `for`, ... are all implemented using custom tags - a main difference to other templating languages.

Some benefits of this approach are:

- You cannot override already existing view helpers by accident.
- It is very easy to write custom view helpers, which live next to the standard view helpers

- All user documentation for a view helper can be automatically generated from the annotations and code documentation. This includes Eclipse autocompletion

**Footnote** This is done through XML Schema Definition files which are generated from the view helper's PHPdoc comments.

Most view helpers have some parameters. These can be plain strings, just like in `<f:link.action controller="Administration">...</f:link.action>`, but as well arbitrary objects. Parameters of view helpers will just be parsed with the same rules as the rest of the template, thus you can pass arrays or objects as parameters.

This is often used when adding arguments to links:

#### ▼ *Creating a link with arguments*

```
<f:link.action controller="Blog" action="show" arguments="{id : blogPost.id}">... read
```

Here, the view helper will get a parameter called `arguments` which is of type array.

**Warning** Make sure you *do not put a space* before or after the opening or closing brackets of an array. If you type `arguments=" {id : blogPost.id}"` (notice the space before the opening curly bracket), the array is automatically casted to a string (as a string concatenation takes place).

This also applies when using object accessors: `<f:do.something with="{object}" />` and `<f:do.something with=" {object}" />` are substantially different: In the first case, the view helper will receive an *object* as argument, while in the second case, it will receive a *string* as argument.

This might first seem like a bug, but actually it is just consistent that it works that way.

#### ▼ **1.2.1 Boolean expressions**

Often, you need some kind of conditions inside your template. For them, you will usually use the `<f:if>` ViewHelper. Now let's imagine we have a list of blog postings and want to display some additional information for the currently selected blog posting. We assume that the currently selected blog is available in `{currentBlogPosting}`. Now, let's have a look how this works:

#### ▼ *Using boolean expressions*

```
<f:for each="{blogPosts}" as="post">
  <f:if condition="{post} == {currentBlogPosting}">... some special output here ...</f:
</f:for>
```

In the above example, there is a bit of new syntax involved: `{post} == {currentBlogPosting}`. Intuitively, this says "if the post I'm currently iterating over is the same as `currentBlogPosting`, do something."

Why can we use this boolean expression syntax? Well, because the `IfViewHelper` has registered the argument `condition` as `boolean`. Thus, the boolean expression syntax is available in all arguments of ViewHelpers which are of type `boolean`.

All boolean expressions have the form *XX Comparator YY*, where:

- *Comparator* is one of the following: `==`, `>`, `>=`, `<`, `<=`, `%` (modulo)
- *XX / YY* is one of the following:
  - A number (integer or float)

- A JSON Array
- A ViewHelper
- An Object Accessor (this is probably the most used example)

### ▼ 1.3 Arrays

Some view helpers, like the `SelectViewHelper` (which renders an HTML select dropdown box), need to get associative arrays as arguments (mapping from internal to displayed name). See the following example how this works:

```
<f:form.select options="{edit: 'Edit item', delete: 'Delete item'}" />
```

The array syntax used here is very similar to the JSON object syntax

**Footnote** Actually, it should be the same. If not, please tell us!

. Thus, the left side of the associative array is used as key without any parsing, and the right side can be either:

- a number

```
{a : 1,
 b : 2
}
```

- a string; Needs to be in either single- or double quotes. In a double-quoted string, you need to escape the " with a \ in front (and vice versa for single quoted strings).

```
{a : 'Hallo',
 b : "Second string with escaped \" (double quotes) but not escaped ' (single quote)
}
```

- a nested array

```
{a : {
  a1 : "bla1",
  a2 : "bla2"
},
 b : "hallo"
}
```

- a variable reference (=an object accessor)

```
{blogTitle : blog.title,
 blogObject: blog
}
```

### ▼ 2 Passing data to the view

You can pass arbitrary objects to the view, using `$this->view->assign($IdentifierString, $Object)` from within the controller. See the above paragraphs about Object Accessors for details how to use the passed data.

### ▼ 3 Layouts

In almost all web applications, there are many similarities between each page. Usually, there are common

templates or menu structures which will not change for many pages.

To make this possible in Fluid, we created a layout system, which we will introduce in this section.

### 3.1 Writing a layout

Every layout is placed in the `Resources/Private/Layouts` directory, and has the file ending `.html`. A layout is a normal Fluid template file, except there are some parts where the actual content of the target page should be inserted.

#### An example layout

```
<html>
<head><title>My fancy web application</title></head>
<body>
<div id="menu">... menu goes here ...</div>
<div id="content">
  <f:render section="content" />
</div>
</body>
</html>
```

With this tag, a section from the target template is rendered.

### 3.2 Using a layout

Using a layout involves two steps:

- Declare which layout to use: `<f:layout name="..." />` can be written anywhere on the page (though we suggest to write it on top, right after the namespace declaration) - the given name references the layout.
- Provide the content for all sections used by the layout using the `<f:section>...</f:section>` tag:

```
<f:section name="content">...</f:section>
```

For the above layout, a minimal template would look like the following:

#### A template for the above layout

```
<f:layout name="example.html" />

<f:section name="content">
  This HTML here will be outputted to inside the layout
</f:section>
```

## 4 Writing your own View Helper

As we have seen before, *all output logic resides in View Helpers*. This includes the standard control flow operators such as `if/else`, HTML forms, and much more. This is the concept which makes Fluid extremely versatile and extensible.

If you want to create a view helper which you can call from your template (as a tag), you just write a plain PHP class which needs to inherit from `F3\Fluid\Core\AbstractViewHelper` (or its subclasses). You need to implement only one method to write a view helper:

```
public function render()
```

## ▼ 4.1 Rendering the View Helper

We refresh what we have learned so far: When a user writes something like `<blog:displayNews />` inside a template (and has imported the "blog" namespace to `F3\Blog\ViewHelpers`), Fluid will automatically instantiate the class `F3\Blog\ViewHelpers\DisplayNewsViewHelper`, and invoke the `render()` method on it.

This `render()` method should return the rendered content as string.

You have the following possibilities to access the environment when rendering your view helper:

- `$this->arguments` is a read-only associative array where you will find the values for all arguments you registered previously.
- `$this->renderChildren()` renders everything between the opening and closing tag of the view helper and returns the rendered result (as string).
- `$this->variableContainer` is an instance of `F3\Fluid\Core\ViewHelper\TemplateVariableContainer`, with which you have access to all variables currently available in the template.

Additionally, you can add variables to the container with

`$this->variableContainer->add($identifier, $value)`, but you have to make sure that you *remove every variable you added* again! This is a security measure against side-effects.

It is also not possible to add a variable to the VariableContainer if a variable of the same name already exists - again to prevent side effects and scope problems.

Now, we will look at an example: How to write a view helper giving us the `foreach` functionality of PHP.

**Footnote** This view helper is already available in the standard library as `<f:for>..</f:for>`. We still use it as example here, as it is quite simple and shows many possibilities.

### ▼ Implementing a loop

A loop could be called within the template in the following way:

```
<f:for each="{blogPosts}" as="blogPost">
  <h2>{blogPost.title}</h2>
</f:for>
```

So, in words, what should the loop do?

It needs two arguments:

- `each`: Will be set to some *object*

**Footnote** Remember that the view helper can receive arbitrary objects as parameters!

which can be iterated over.

- `as`: The *name* of a variable which will contain the current element being iterated over

It then should do the following (in pseudocode):

```
foreach ($each as $$as) {
  // render everything between opening and closing tag
}
```

Implementing this is fairly straightforward, as you will see right now:

```

class ForViewHelper {
    /**
     * Renders a loop
     *
     * @param array $each Array to iterate over❶
     * @param string $as Iteration variable
     */
    public function render(array $each, $as) {❷
        $out = '';
        foreach ($each as $singleElement) {
            $this->variableContainer->add($as, $singleElement);
            $out .= $this->renderChildren();❸
            $this->variableContainer->remove($as);
        }
        return $out;
    }
}

```

- ❶ The PHPDoc *is part of the code!* Fluid extracts the argument datatypes from the PHPDoc.
- ❷ You can simply register arguments to the view helper by adding them as method arguments of the `render()` method.
- ❸ Here, everything between the opening and closing tag of the view helper is rendered and returned as string.

The above example demonstrates how we add a variable, render all children (everything between the opening and closing tag), and remove the variable again to prevent side-effects.

## 4.2 Declaring arguments

We have now seen that we can add arguments just by adding them as method arguments to the `render()` method. There is, however, a second method to register arguments:

You can also register arguments inside a method called `initializeArguments()`. Call

```
$this->registerArgument($name, $dataType, $description, $isRequired, $defaultValue=NULL)
```

inside.

It depends how many arguments a view helper has. Sometimes, registering them as `render()` arguments is more beneficial, and sometimes it makes more sense to register them in `initializeArguments()`.

## 4.3 TagBasedViewHelper

Many view helpers output an HTML tag - for example `<f3:link.action ...>` outputs a `<a href="...">` tag. There are many view helpers which work that way.

Very often, you want to add a CSS class or a target attribute to an `<a href="...">` tag. This often leads to repetitive code like below. (Don't look at the code too thoroughly, it should just demonstrate the boring and repetitive task one would have without the `TagBasedViewHelper`).

```

class LinkViewHelper extends \F3\Fluid\Core\AbstractViewHelper {
    public function initializeArguments() {

```

```

    $this->registerArgument('class', 'string', 'CSS class to add to the link');
    $this->registerArgument('target', 'string', 'Target for the link');
    ... and more ...
}
public function render() {
    $output = '<a href="..."';
    if ($this->arguments['class']) {
        $output .= ' class="' . $this->arguments['class'] . '"';
    }
    if ($this->arguments['target']) {
        $output .= ' target="' . $this->arguments['target'] . '"';
    }
    $output .= '>';
    ... and more ...
    return $output;
}
}

```

Now, the `TagBasedViewHelper` introduces two more methods you can use inside `initializeArguments()`:

- `registerTagAttribute($name, $type, $description, $required)`: Use this method to register an attribute which should be directly added to the tag
- `registerUniversalTagAttributes()`: If called, registers the standard HTML attributes (class, id, dir, lang, style, title).

Inside the `TagBasedViewHelper`, there is a `TagBuilder` object available (with `$this->tag`) which makes building a tag a lot more straightforward

With the above methods we get, the `LinkViewHelper` from above can be condensed as follows:

```

class LinkViewHelper extends \F3\Fuild\Core\AbstractViewHelper {
    public function initializeArguments() {
        $this->registerUniversalTagAttributes();
    }

    /**
     * Render the link.
     *
     * @param string $action Target action
     * @param array $arguments Arguments
     * @param string $controller Target controller. If NULL current controllerName is used
     * @param string $package Target package. if NULL current package is used
     * @param string $subpackage Target subpackage. if NULL current subpackage is used
     * @param string $section The anchor to be added to the URI
     * @return string The rendered link
     */
    public function render($action = NULL, array $arguments = array(), $controller = NULL,
        $uriBuilder = $this->controllerContext->getURIBuilder();
        $uri = $uriBuilder->URIFor($action, $arguments, $controller, $package, $subpackage, $

```

```

$this->tag->addAttribute('href', $uri);
$this->tag->setContent($this->renderChildren());

return $this->tag->render();
}
}

```

Additionally, we now already have support for all universal HTML attributes.

You might now think that the building blocks are ready, but there is one more nice thing to add: `additionalAttributes`! Read about it in the next section.

#### ▼ 4.3.1 `additionalAttributes`

Sometimes, you need some HTML attributes which are not part of the standard. As an example: if you use the Dojo JavaScript framework, using these non-standard attributes makes life a lot easier.

**Footnote** There are always some religious discussions whether to allow non-standard attributes or not. People being against it argue that it "pollutes" HTML, and makes it not validate anymore. More pragmatic people see some benefits to custom attributes in some contexts: If you use JavaScript to evaluate them, they will be ignored by the rendering engine if JavaScript is switched off, and can enable special behavior when JavaScript is turned on. Thus, they can make it easy to provide degradable interfaces.

(Before bashing Dojo now: Of course you do not *need* the additional HTML arguments, but they make work with it a lot more comfortable)

We think that the templating framework should not constrain the user in his possibilities - thus, it should be possible to add custom HTML attributes as well, if they are needed (People who have already worked with JSP know that it can be difficult to archive this. Our solution looks as follows:

*Every view helper which inherits from `TagBasedViewHelper` has a special property called `additionalAttributes` which allows you to add arbitrary HTML attributes to the tag.*

`additionalAttributes` should be an associative array, where the key is the name of the HTML attribute.

If the link tag from above needed a new attribute called `fadeDuration`, which is not part of HTML, you could do that as follows:

```
<f:link.action ... additionalAttributes="{fadeDuration : 800}">Link with fadeDuration s
```

This attribute is available in all tags that inherit from `F3\Fluid\Core\ViewHelper\TagBasedViewHelper`.

## ▼ 4.4 Facets

The possibilities you get when you base your view helper on

`F3\Fluid\Core\ViewHelper\AbstractViewHelper` should be enough for most use cases - however, there are some cases when the view helper needs to interact in a special way with its surroundings - an example is the "if/else" view helper group.

If a view helper needs to know more about its surroundings, it has to implement a certain facet. Facets are plain PHP interfaces.

*Currently, all facets are NOT YET PUBLIC API! Use them at your own risk!*

### ▼ 4.4.1 `SubNodeAccess Facet`

Sometimes, a view helper needs direct access to its child nodes - as it does not want to render all of its children, but only a subset. For this to work the `SubNodeAccessInterface` has been introduced.

Let's take `if/then/else` as an example and start with two examples how this view helper is supposed to work:

```
<f:if condition="...">
    This text should only be rendered if the condition evaluates to TRUE.
</f:if>
```

This above case is the most simple case. However, we want to support `if/else` as well:

```
<f:if condition="...">
    <f:then>If condition evaluated to TRUE, "then" should be rendered</f:then>
    <f:else>If condition evaluated to FALSE, "else" should be rendered</f:else>
</f:if>
```

To implement the functionality of the `<f:if>` view helper, a standard `$this->renderChildren()` will not be sufficient, as the `if`-Tag has no control whether the `<f:then>` or `<f:else>` is rendered. Thus, the `<f:if>` tag needs more information about its environment, namely it needs access to its subnodes in the syntax tree.

To make this work, the `<f:if>`-tag implements the `F3\Fluid\Core\Facets\SubNodeAccessInterface`. Now, the method `setChildren(array $childNodes)` (defined in the interface) will be called before the `render()` method is invoked. Thus, the view helper has all of its subnodes directly available in the `render()` method and can decide which subnodes it will render based on arbitrary conditions.

#### ▼ 4.4.2 PostParse Facet

**Note** This facet will be only needed in exceptional cases, so before you use it, try to think of a different way to do it. Using this facet can easily break template parsing if you do not know what you are doing.

Sometimes, the presence of a tag affects global rendering behavior - as seen in the template/layout subsystem: With the tag `<f:layout name="..." />` the user can specify a layout name for the current template. Somehow, the parser needs to know if a layout was selected directly after parsing the template - before any data has been passed to it.

Thus, if a view helper implements the `F3\Fluid\Core\ViewHelper\Facets\PostParseInterface`, it can specify a callback which is called *directly after the tag has been parsed in the template*. The method signature looks as follows:

```
static public function postParseEvent(\F3\Fluid\Core\SyntaxTree\ViewHelperNode $syntaxT
```

Note this method is *static*

**Footnote** It is static because the `ViewHelper` has not been instantiated at that point in time, thus it is forbidden to set any instance variables.

. The arguments the method receives are as follows:

- A reference to the current syntax tree node (which is always a `ViewHelperNode`). This is particularly useful if a `ViewHelper` wants to store a reference to its node in the `variableContainer`.
- The view helper arguments. This is an associative array, with the argument name as key, and the associated syntax tree (an instance of `F3\Fluid\Core\SyntaxTree\RootNode`) as value. Because variables are not bound at this point, you always need to call `evaluate(...)` on the arguments you want to receive. Look into the `LayoutViewHelper` for an example.

- A parsing variable container. *This is not the VariableContainer used for rendering!* The supplied VariableContainer is initially empty, and is used to pass data from ViewHelpers to the View. It is used mainly in the LayoutViewHelper and SectionViewHelper.

## ▼ 5 Standard View Helper Library

Should be autogenerated from the tags.

### ▼ 5.1 base

View helper which creates a `<base href="..."></base>` tag.

#### ▼ 5.1.1 Examples

##### ▼ Example

```
<f:base />
```

Output:

```
<base href="http://yourdomain.tld/"></base>
```

(depending on your domain)

#### ▼ 5.1.2 Arguments

No arguments defined.

### ▼ 5.2 else

"ELSE" -> only has an effect inside of "IF". See If-ViewHelper for documentation.

#### ▼ 5.2.1 Arguments

No arguments defined.

### ▼ 5.3 for

Loop view helper

#### ▼ 5.3.1 Examples

##### ▼ Simple

```
<f:for each="{0:1, 1:2, 2:3, 3:4}" as="foo">{foo}</f:for>
```

Output:

1234

##### ▼ Output array key

```
<ul>
  <f:for each="{fruit1: 'apple', fruit2: 'pear', fruit3: 'banana', fruit4: 'cherry'}"
    <li>{label}: {fruit}</li>
  </f:for>
</ul>
```

Output:

```

<ul>
<li>fruit1: apple</li>
<li>fruit2: pear</li>
<li>fruit3: banana</li>
<li>fruit4: cherry</li>
</ul>

```

### 5.3.2 Arguments

#### Arguments

| Name    | Type    | Required | Description   | Default |
|---------|---------|----------|---|---------|
| each    | array   | yes      | The array to be iterated over                           |         |
| as      | string  | yes      | The name of the iteration variable                      |         |
| key     | string  | no       | The name of the variable to store the current array key |         |
| reverse | boolean | no       | If enabled, the iterator runs reversely.                | FALSE   |

### 5.4 form

Form view helper. Generates a `<form>` Tag.

#### 5.4.1 Basic usage

Use `<f:form>` to output an HTML `<form>` tag which is targeted at the specified action, in the current controller and package.

It will submit the form data via a POST request. If you want to change this, use `method="get"` as an argument.

#### Example

```
<f:form action="...">...</f:form>
```

#### 5.4.2 A complex form with a specified encoding type

##### Form with enctype set

```
<f:form action=".." controllerName="..." packageName="..." enctype="multipart/form-data">
```

#### 5.4.3 A Form which should render a domain object

##### Binding a domain object to a form

```

<f:form action="..." name="customer" object="{customer}">
  <f:form.hidden property="id" />
  <f:form.textbox property="name" />

```

```
</f:form>
```

This automatically inserts the value of {customer.name} inside the textbox and adjusts the name of the textbox accordingly.

#### ▼ 5.4.4 Arguments

##### ▼ Arguments

| Name                 | Type   | Required | Description  | Default |
|----------------------|--------|----------|--|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag.            |         |
| action               | string | no       | target action  |         |
| arguments            | array  | no       | additional arguments   | Array   |
| controllerName       | string | no       | target controller  |         |
| packageName          | string | no       | target package   |         |
| subpackageName       | string | no       | target subpackage  |         |
| object               | mixed  | no       | object to use for the form. Use in conjunction with the "property" attribute on the sub tags |         |
| section              | string | no       | The anchor to be added to the URI  |         |
| enctype              | string | no       | MIME type with which the form is submitted   |         |
| method               | string | no       | Transfer type (GET or POST)  |         |
| name                 | string | no       | Name of form   |         |
| onreset              | string | no       | JavaScript: On reset of the form   |         |
| onsubmit             | string | no       | JavaScript: On submit of the form  |         |
| class                | string | no       | CSS class(es) for this element   |         |
| dir                  | string | no       | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl"       |         |

|           |         |    | (right to left)  |  |
|-----------|---------|----|--|--|
| id        | string  | no | Unique (in this file) identifier for this HTML element.          |  |
| lang      | string  | no | Language for this element. Use short names specified in RFC 1766 |  |
| style     | string  | no | Individual CSS styles for this element                           |  |
| title     | string  | no | Tooltip text of element  |  |
| accesskey | string  | no | Keyboard shortcut to access this element                         |  |
| tabindex  | integer | no | Specifies the tab order of this element                          |  |

## 5.5 form.hidden

Renders an `<input type="hidden" ...>` tag.

### 5.5.1 Examples

#### Example

```
<f:hidden name="myHiddenValue" value="42" />
```

Output:

```
<input type="hidden" name="myHiddenValue" value="42" />
```

You can also use the "property" attribute if you have bound an object to the form.

See `<f:form>` for more documentation.

### 5.5.2 Arguments

#### Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag. |         |
| name                 | string | no       | Name of input tag   |         |
| value                | mixed  | no       | Value of input tag  |         |
| property             | string | no       | Name of Object  |         |

|           |         |    |  |  |
|-----------|---------|----|--|--|
|           |         |    | Property. If used in conjunction with <f3:form object="...">, "name" and "value" properties will be ignored. |  |
| class     | string  | no | CSS class(es) for this element   |  |
| dir       | string  | no | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl" (right to left)       |  |
| id        | string  | no | Unique (in this file) identifier for this HTML element.  |  |
| lang      | string  | no | Language for this element. Use short names specified in RFC 1766   |  |
| style     | string  | no | Individual CSS styles for this element   |  |
| title     | string  | no | Tooltip text of element  |  |
| accesskey | string  | no | Keyboard shortcut to access this element   |  |
| tabindex  | integer | no | Specifies the tab order of this element  |  |

## ▼ 5.6 form.select

This view helper generates a <select> dropdown list for the use with a form.

### ▼ 5.6.1 Basic usage

The most straightforward way is to supply an associative array as the "options" parameter.

The array key is used as option key, and the value is used as human-readable name.

#### ▼ Basic usage

```
<f3:form.select name="paymentOptions" options="{paypal: 'PayPal International Services'
```

### ▼ 5.6.2 Pre-select a value

To pre-select a value, set "selectedValue" to the option key which should be selected.

#### ▼ Default value

```
<f3:form.select name="paymentOptions" options="{paypal: 'PayPal International Services'
```

Generates a dropdown box like above, except that "VISA Card" is selected.

If the select box is a multi-select box (multiple="true"), then "selectedValue" can be an array as well.

### 5.6.3 Usage on domain objects

If you want to output domain objects, you can just pass them as array into the "options" parameter.

To define what domain object value should be used as option key, use the "optionValueField" variable. Same goes for optionLabelField.

If the optionValueField variable is set, the getter named after that value is used to retrieve the option key.

If the optionLabelField variable is set, the getter named after that value is used to retrieve the option value.

#### Domain objects

```
<f3:form.select name="users" options="{userArray}" optionValueField="id" optionLabelField="firstName">
```

In the above example, the userArray is an array of "User" domain objects, with no array key specified.

So, in the above example, the method \$user->getId() is called to retrieve the key, and \$user->getFirstName() to retrieve the displayed value of each entry.

The "selectedValue" property now expects a domain object, and tests for object equivalence.

### 5.6.4 Arguments

#### Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag.   |         |
| name                 | string | no       | Name of input tag   |         |
| value                | mixed  | no       | Value of input tag  |         |
| property             | string | no       | Name of Object Property. If used in conjunction with <f3:form object="...">, "name" and "value" properties will be ignored. |         |
| class                | string | no       | CSS class(es) for this element  |         |
| dir                  | string | no       | Text direction for this HTML element.<br>Allowed strings: "ltr"   |         |

|                  |         |    |  |  |
|------------------|---------|----|--|--|
|                  |         |    | (left to right), "rtl"<br>(right to left)  |  |
| id               | string  | no | Unique (in this file) identifier for this HTML element.                                    |  |
| lang             | string  | no | Language for this element. Use short names specified in RFC 1766                           |  |
| style            | string  | no | Individual CSS styles for this element   |  |
| title            | string  | no | Tooltip text of element  |  |
| accesskey        | string  | no | Keyboard shortcut to access this element   |  |
| tabindex         | integer | no | Specifies the tab order of this element  |  |
| multiple         | string  | no | if set, multiple select field  |  |
| size             | string  | no | Size of input field  |  |
| options          | array   | no | Associative array with internal IDs as key, and the values are displayed in the select box |  |
| optionValueField | string  | no | If specified, will call the appropriate getter on each object to determine the value.      |  |
| optionLabelField | string  | no | If specified, will call the appropriate getter on each object to determine the label.      |  |

## ▼ 5.7 form.submit

Creates a submit button.

### ▼ 5.7.1 Examples

#### ▼ Defaults

```
<f:submit value="Send Mail" />
```

Output:

<input type="submit" />

▼ *Dummy content for template preview*

```
<f:submit name="mySubmit" value="Send Mail"><button>dummy button</button></f:submit>
```

Output:

```
<input type="submit" name="mySubmit" value="Send Mail" />
```

▼ **5.7.2 Arguments**

▼ *Arguments*

| Name                 | Type    | Required | Description  | Default |
|----------------------|---------|----------|--|---------|
| additionalAttributes | array   | no       | Additional tag attributes. They will be added directly to the resulting HTML tag.                      |         |
| name                 | string  | no       | Name of submit tag   |         |
| value                | string  | no       | Value of submit tag  |         |
| class                | string  | no       | CSS class(es) for this element   |         |
| dir                  | string  | no       | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl" (right to left) |         |
| id                   | string  | no       | Unique (in this file) identifier for this HTML element.  |         |
| lang                 | string  | no       | Language for this element. Use short names specified in RFC 1766                                       |         |
| style                | string  | no       | Individual CSS styles for this element   |         |
| title                | string  | no       | Tooltip text of element  |         |
| accesskey            | string  | no       | Keyboard shortcut to access this element   |         |
| tabindex             | integer | no       | Specifies the tab order of this element  |         |

## ▼ 5.8 form.textarea

Textarea view helper.

The value of the text area needs to be set via the "value" attribute, as with all other form ViewHelpers.

### ▼ 5.8.1 Examples

#### ▼ Example

```
<f:textarea name="myTextArea" value="This is shown inside the textarea" />
```

Output:

```
<textarea name="myTextArea">This is shown inside the textarea</textarea>
```

### ▼ 5.8.2 Arguments

#### ▼ Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag.   |         |
| name                 | string | no       | Name of input tag   |         |
| value                | mixed  | no       | Value of input tag  |         |
| property             | string | no       | Name of Object Property. If used in conjunction with <f3:form object="...">, "name" and "value" properties will be ignored. |         |
| rows                 | int    | yes      | The number of rows of a text area   |         |
| cols                 | int    | yes      | The number of columns of a text area  |         |
| class                | string | no       | CSS class(es) for this element  |         |
| dir                  | string | no       | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl" (right to left)                      |         |
| id                   | string | no       | Unique (in this file)   |         |

|           |         |    |  |  |
|-----------|---------|----|--|--|
|           |         |    | identifier for this HTML element.                                |  |
| lang      | string  | no | Language for this element. Use short names specified in RFC 1766 |  |
| style     | string  | no | Individual CSS styles for this element                           |  |
| title     | string  | no | Tooltip text of element  |  |
| accesskey | string  | no | Keyboard shortcut to access this element                         |  |
| tabindex  | integer | no | Specifies the tab order of this element                          |  |

## ▼ 5.9 form.textbox

View Helper which creates a simple Text Box (<input type="text">).

### ▼ 5.9.1 Examples

#### ▼ Example

```
<f:textbox name="myTextBox" value="default value" />
```

Output:

```
<input type="text" name="myTextBox" value="default value" />
```

### ▼ 5.9.2 Arguments

#### ▼ Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag.   |         |
| name                 | string | no       | Name of input tag   |         |
| value                | mixed  | no       | Value of input tag  |         |
| property             | string | no       | Name of Object Property. If used in conjunction with <f3:form object="...">, "name" and "value" properties will be ignored. |         |

|           |         |    |  |  |
|-----------|---------|----|--|--|
| class     | string  | no | CSS class(es) for this element   |  |
| dir       | string  | no | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl" (right to left) |  |
| id        | string  | no | Unique (in this file) identifier for this HTML element.  |  |
| lang      | string  | no | Language for this element. Use short names specified in RFC 1766                                       |  |
| style     | string  | no | Individual CSS styles for this element   |  |
| title     | string  | no | Tooltip text of element  |  |
| accesskey | string  | no | Keyboard shortcut to access this element   |  |
| tabindex  | integer | no | Specifies the tab order of this element  |  |

## ▼ 5.10 form.upload

A view helper which generates an `<input type="file">` HTML element.

Make sure to set `enctype="multipart/form-data"` on the form!

### ▼ 5.10.1 Examples

#### ▼ Example

```
<f:upload name="file" />
```

Output:

```
<input type="file" name="file" />
```

### ▼ 5.10.2 Arguments

#### ▼ Arguments

| Name                 | Type  | Required | Description  | Default |
|----------------------|-------|----------|--|---------|
| additionalAttributes | array | no       | Additional tag attributes. They will be added directly to the resulting HTML |         |

|           |         |    | tag.  |  |
|-----------|---------|----|---|--|
| name      | string  | no | Name of input tag   |  |
| value     | mixed   | no | Value of input tag  |  |
| property  | string  | no | Name of Object Property. If used in conjunction with <f3:form object="...">, "name" and "value" properties will be ignored. |  |
| class     | string  | no | CSS class(es) for this element  |  |
| dir       | string  | no | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl" (right to left)                      |  |
| id        | string  | no | Unique (in this file) identifier for this HTML element.   |  |
| lang      | string  | no | Language for this element. Use short names specified in RFC 1766  |  |
| style     | string  | no | Individual CSS styles for this element  |  |
| title     | string  | no | Tooltip text of element   |  |
| accesskey | string  | no | Keyboard shortcut to access this element  |  |
| tabindex  | integer | no | Specifies the tab order of this element   |  |

## ▼ 5.11 format.crop

Use this view helper to crop the text between its opening and closing tags.

### ▼ 5.11.1 Examples

#### ▼ Defaults

```
<f:format.crop maxCharacters="10">This is some very long text</f:format.crop>
```

Output:

This is so...

▼ *Custom suffix*

```
<f:format.crop maxCharacters="17" append=" [more]">This is some very long text</f:format.crop>
```

Output:

This is some very [more]

WARNING: This tag does NOT handle tags currently.

WARNING: This tag doesn't care about multibyte charsets currently.

▼ **5.11.2 Arguments**

▼ *Arguments*

| Name          | Type    | Required | Description                            | Default |
|---------------|---------|----------|--|---------|
| maxCharacters | integer | yes      | Place where to truncate the string     |         |
| append        | string  | no       | What to append, if truncation happened | ...     |

▼ **5.12 format.currency**

Formats a given float to a currency representation.

▼ **5.12.1 Examples**

▼ *Defaults*

```
<f:format.currency>123.456</f:format.currency>
```

Output:

123,46

▼ *All parameters*

```
<f:format.currency currencySign="$" decimalSeparator="." thousandsSeparator=",">54321</f:format.currency>
```

Output:

54,321.00 \$

▼ **5.12.2 Arguments**

▼ *Arguments*

| Name             | Type   | Required | Description                                     | Default |
|------------------|--------|----------|---|---------|
| currencySign     | string | no       | (optional) The currency sign, eg \$ or €.       |         |
| decimalSeparator | string | no       | (optional) The separator for the decimal point. | ,       |

|                    |        |    |                                     |   |
|--------------------|--------|----|-------------------------------------|---|
| thousandsSeparator | string | no | (optional) The thousands separator. | . |
|--------------------|--------|----|-------------------------------------|---|

## ▼ 5.13 format.date

Formats a \DateTime object.

### ▼ 5.13.1 Examples

#### ▼ Defaults

```
<f:format.date>{dateObject}</f:format.date>
```

Output:

1980-12-13

(depending on the current date)

#### ▼ Custom date format

```
<f:format.date format="H:i">{dateObject}</f:format.date>
```

Output:

01:23

(depending on the current time)

#### ▼ strtotime string

```
<f:format.date format="d.m.Y - H:i:s">+1 week 2 days 4 hours 2 seconds</f:format.date>
```

Output:

13.12.1980 - 21:03:42

(depending on the current time, see <http://www.php.net/manual/en/function.strftime.php>)

### ▼ 5.13.2 Arguments

#### ▼ Arguments

| Name   | Type   | Required | Description  | Default |
|--------|--------|----------|--|---------|
| format | string | no       | Format String which is taken to format the Date/Time | Y-m-d   |

## ▼ 5.14 format.nl2br

Wrapper for PHP's nl2br function.

### ▼ 5.14.1 Arguments

No arguments defined.

## ▼ 5.15 format.number

Formats a number with custom precision, decimal point and grouped thousands.

### 5.15.1 Arguments

#### Arguments

| Name               | Type   | Required | Description                                    | Default |
|--------------------|--------|----------|--|---------|
| decimals           | int    | no       | The number of digits after the decimal point   | 2       |
| decimalSeparator   | string | no       | The decimal point character                    | .       |
| thousandsSeparator | string | no       | The character for grouping the thousand digits | ,       |

### 5.16 format.printf

A view helper for formatting values with printf. Either supply an array for the arguments or a single value.

See <http://www.php.net/manual/en/function.sprintf.php>

#### 5.16.1 Examples

##### Scientific notation

```
<f:format.printf arguments="{number : 362525200}">%.3e</f:format.p
```

Output:

3.625e+8

##### Argument swapping

```
<f:format.printf arguments="{0: 3,1: 'Kasper'}">%2$s is great, TYPO%1$d too. Yes, TYPO3
```

Output:

Kasper is great, TYPO3 too. Yes, TYPO3 is great and so is Kasper!

##### Single argument

```
<f:format.printf arguments="{1:'TYPO3'}">We love %s</f:format.p
```

Output:

We love TYPO3

#### 5.16.2 Arguments

##### Arguments

| Name      | Type  | Required | Description                | Default |
|-----------|-------|----------|----------------------------|---------|
| arguments | array | yes      | The arguments for vsprintf |         |

### 5.17 if

This view helper implements an if/else condition.

### 5.17.1 Arguments

#### Arguments

| Name      | Type    | Required | Description           | Default |
|-----------|---------|----------|-----------------------|---------|
| condition | boolean | yes      | View helper condition |         |

### 5.18 layout

With this tag, you can select a layout to be used.

#### 5.18.1 Arguments

#### Arguments

| Name | Type   | Required | Description  | Default |
|------|--------|----------|--|---------|
| name | string | yes      | Name of layout to use. If none given, "default" is used. |         |

### 5.19 link.action

A view helper for creating links to actions.

#### 5.19.1 Examples

▼ D

```
<f:link.action>some link</f:link.action>
```

Output:

```
<a href="currentpackage/currentcontroller">some link</a>
```

(depending on routing setup and current package/controller/action)

#### Additional arguments

```
<f:link.action action="myAction" controller="MyController" package="MyPackage" subpackage="MySubpackage">some link</f:link.action>
```

Output:

```
<a href="mypackage/mycontroller/mysubpackage/myaction?key1=value1&key2=value2">some link</a>
```

(depending on routing setup)

#### 5.19.2 Arguments

#### Arguments

| Name                 | Type  | Required | Description   | Default |
|----------------------|-------|----------|---|---------|
| additionalAttributes | array | no       | Additional tag attributes. They will be added directly to |         |

|            |         |    | the resulting HTML tag.  |       |
|------------|---------|----|--|-------|
| action     | string  | no | Target action  |       |
| arguments  | array   | no | Arguments  | Array |
| controller | string  | no | Target controller. If NULL current controllerName is used  |       |
| package    | string  | no | Target package. if NULL current package is used  |       |
| subpackage | string  | no | Target subpackage. if NULL current subpackage is used  |       |
| section    | string  | no | The anchor to be added to the URI  |       |
| format     | string  | no | The format to render   | html  |
| class      | string  | no | CSS class(es) for this element   |       |
| dir        | string  | no | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl" (right to left) |       |
| id         | string  | no | Unique (in this file) identifier for this HTML element.  |       |
| lang       | string  | no | Language for this element. Use short names specified in RFC 1766                                       |       |
| style      | string  | no | Individual CSS styles for this element   |       |
| title      | string  | no | Tooltip text of element  |       |
| accesskey  | string  | no | Keyboard shortcut to access this element   |       |
| tabindex   | integer | no | Specifies the tab order of this element  |       |

## ▼ 5.20 link.email

Email link view helper.

Generates an email link.

### ▼ 5.20.1 Examples

```
<code title="basic email link">
```

### ▼ 5.20.2 Arguments

#### ▼ Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag. |         |
| email                | string | yes      | The email address to be turned into a link.                                       |         |

## ▼ 5.21 link.external

A view helper for creating links to external targets.

### ▼ 5.21.1 Examples

#### ▼ Example

```
<f:link.external uri="http://www.typo3.org" target="_blank">external link</f:link.external>
```

Output:

```
<a href="http://www.typo3.org" target="_blank">external link</a>
```

### ▼ 5.21.2 Arguments

#### ▼ Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag. |         |
| uri                  | string | yes      | the URI that will be put in the href attribute of the rendered link tag           |         |
| class                | string | no       | CSS class(es) for this element  |         |

|           |         |    |  |  |
|-----------|---------|----|--|--|
| dir       | string  | no | Text direction for this HTML element.<br>Allowed strings: "ltr" (left to right), "rtl" (right to left) |  |
| id        | string  | no | Unique (in this file) identifier for this HTML element.  |  |
| lang      | string  | no | Language for this element. Use short names specified in RFC 1766                                       |  |
| style     | string  | no | Individual CSS styles for this element   |  |
| title     | string  | no | Tooltip text of element  |  |
| accesskey | string  | no | Keyboard shortcut to access this element   |  |
| tabindex  | integer | no | Specifies the tab order of this element  |  |
| target    | string  | no | Target of link   |  |
| rel       | string  | no | Specifies the relationship between the current document and the linked document                        |  |

## ▼ 5.22 render

### ▼ 5.22.1 Arguments

#### ▼ Arguments

| Name    | Type   | Required | Description   | Default |
|---------|--------|----------|---|---------|
| section | string | no       | Name of section to render. If used in a layout, renders a section of the main content file. If used inside a standard template, renders a section of the same file. |         |
| partial | string | no       | Reference to a  |         |

|           |       |    |                                   |       |
|-----------|-------|----|-----------------------------------|-------|
|           |       |    | partial.                          |       |
| arguments | array | no | Arguments to pass to the partial. | Array |

### ▼ 5.23 section

A Section view helper

#### ▼ 5.23.1 Arguments

##### ▼ Arguments

| Name | Type   | Required | Description         | Default |
|------|--------|----------|---------------------|---------|
| name | string | yes      | Name of the section |         |

### ▼ 5.24 then

"THEN" -> only has an effect inside of "IF". See If-ViewHelper for documentation.

#### ▼ 5.24.1 Arguments

No arguments defined.

### ▼ 5.25 uri.action

A view helper for creating URIs to actions.

#### ▼ 5.25.1 Examples

##### ▼ Def

```
<f:uri.action>some link</f:uri.action>
```

Output:

currentpackage/currentcontroller

(depending on routing setup and current package/controller/action)

##### ▼ Additional arguments

```
<f:uri.action action="myAction" controller="MyController" package="MyPackage" subpackage="MySubpackage">
```

Output:

mypackage/mycontroller/mysubpackage/myaction?key1=value1&key2=value2

(depending on routing setup)

#### ▼ 5.25.2 Arguments

##### ▼ Arguments

| Name       | Type   | Required | Description           | Default |
|------------|--------|----------|-----------------------|---------|
| action     | string | no       | Target action         |         |
| arguments  | array  | no       | Arguments             | Array   |
| controller | string | no       | Target controller. If |         |

|            |         |    |  |       |
|------------|---------|----|--|-------|
|            |         |    | NULL current controllerName is used                              |       |
| package    | string  | no | Target package. if NULL current package is used                  |       |
| subpackage | string  | no | Target subpackage. if NULL current subpackage is used            |       |
| section    | string  | no | The anchor to be added to the URI                                |       |
| format     | string  | no | The format to be added to the URI                                | html  |
| absolute   | boolean | no | If enabled, an absolute URI (including the base URI) is returned | FALSE |

## ▼ 5.26 uri.email

Email link view helper.

Generates an email link.

### ▼ 5.26.1 Examples

```
<code title="basic email link">
```

### ▼ 5.26.2 Arguments

#### ▼ Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag. |         |
| email                | string | yes      | The email address to be turned into a link.                                       |         |

## ▼ 5.27 uri.external

A view helper for creating URIs to external targets.

Currently the specified URI is simply passed through.

### ▼ 5.27.1 Examples

#### ▼ Example

```
<f:uri.external uri="http://www.typo3.org" />
```

Output:

http://www.typo3.org

### ▼ 5.27.2 Arguments

#### ▼ Arguments

| Name                 | Type   | Required | Description   | Default |
|----------------------|--------|----------|---|---------|
| additionalAttributes | array  | no       | Additional tag attributes. They will be added directly to the resulting HTML tag. |         |
| uri                  | string | yes      | the target URI  |         |

## ▼ Chapter 3: Software Design

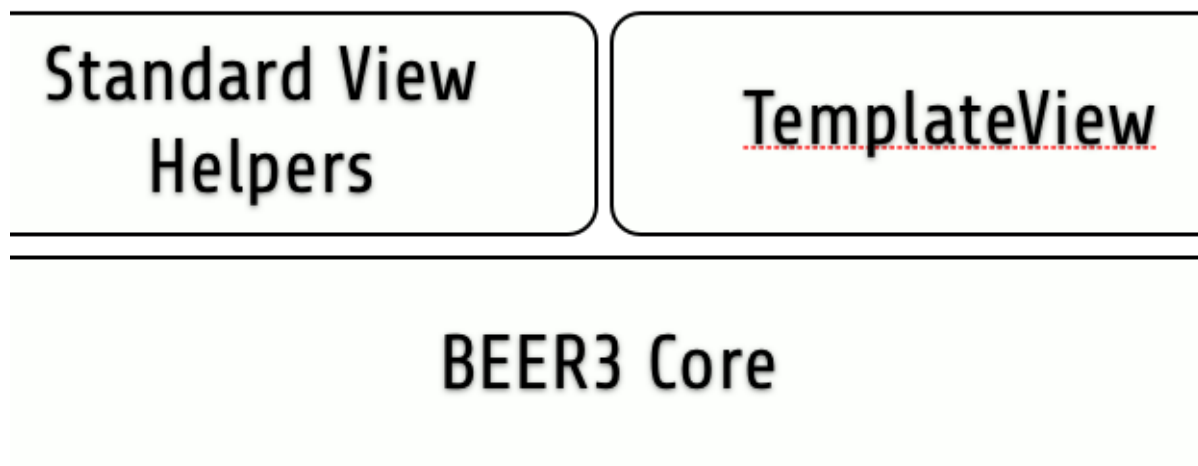
---

This chapter will explain some of the inner workings of Fluid. It is meant for people who want to help developing Fluid, and understand the inner workings of it.

### ▼ 1 Design Decisions

Fluid was born in the context of FLOW3, but from the beginning, we saw the needs for such a templating system in other contexts - namely TYPO3 v4, or stand-alone usage. That's why the core design of Fluid reflects this thinking.

We are using a layered architecture for Fluid, shown below:



Fluid Core consists of the parts which do not change depending on the environment, such as the template parser.

The upper layer, consisting of the Standard View Helpers and the TemplateView, are currently FLOW3-specific, but there will be separate layers for different contexts (like TYPO3 v4) which provide standard view helpers adjusted to the different platforms, different Fluid initialization code, etc.

### ▼ 2 The Core

Fluid Core consists of the following components:

- The `TemplateParser`, which takes a template file and builds up a syntax tree from it.
- The Syntax Tree Elements (which have logic inside them as well)
  - The `AbstractViewHelper`, being the base class for all view helpers.

Rendering a template always involves two steps:

- Call the `parse` method in the `TemplateParser` class, which will return a `SyntaxTree`. This Syntax Tree will be cached in the future.
- The `SyntaxTree` is implemented using a *Composite* design pattern, with all syntax tree nodes extending `\F3\Fluid\Core\SyntaxTree\AbstractNode`.
- Call the `render(\F3\Fluid\Core\VariableContainer $variableContainer)` method. This method needs a `VariableContainer` as argument - this means all bound variables which should be rendered.
- The result of the `render()` method is the output string.

### ▼ 3 The upper layers for FLOW3



## ▼ Chapter 4: Configuration

---

- Technical information; Installation, Reference of TypoScript, configuration options on system level, how to extend it, the technical details, how to debug it.
- Language should be technical, assuming developer knowledge of FLOW3. Small examples/visuals are always encouraged.
- Target group: Developers

## ▼ Chapter 5: Tutorial

---

A full point-a-to-b-to-c walk-through of an application of the package. Include screenshots.