

# **Extbase extension programming**

---

Copyright © 2009 Jochen Rau, Sebastian Kurfuerst, Franz Ripfel

---

**COLLABORATORS**

	<i>TITLE :</i> Extbase extension programming		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Jochen Rau, Sebastian Kurfuerst, and Franz Ripfel	December 10, 2009	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>I Quickstart, get blog_example running and try your own first steps (Franz Ripfel)</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 A new way to write extensions . . . . .	2
1.2 Important terms . . . . .	2
<b>2 Get the official demo blog_example running</b>	<b>3</b>
2.1 Prerequisites: TYPO3 4.3 with pagetree and working basic layout . . . . .	3
2.2 Install extensions from MVC-Project and add plugins to page . . . . .	3
2.3 Check out the functionality of blog_example . . . . .	3
<b>3 Understand the important parts and see how it works</b>	<b>5</b>
3.1 Filestructure and naming conventions . . . . .	5
3.2 How TYPO3 is notified to insert our plugin . . . . .	5
3.3 Inspect the blog controller . . . . .	6
3.4 Inspect the html-templates (and the folder structure they are in) . . . . .	6
3.5 Fluid: Get a basic understanding how the html-templates work . . . . .	7
3.6 Persistence: Or where are the database calls? . . . . .	8
3.6.1 Retrieve specific data (show the blog edit form with the current data of the blog) . . . . .	8
3.6.2 Save / persist data (save/persist the modified data from the form) . . . . .	8
<b>4 Your Turn: Building you own first Extbase-extension</b>	<b>10</b>
4.1 Discuss and decide a Specification . . . . .	10
4.2 Design your model . . . . .	10
4.3 Build your model (part1) . . . . .	11
4.4 Add Controller / View (part1) . . . . .	18
4.5 Expand to whole specification . . . . .	19
4.5.1 Edit existing fridges . . . . .	19
4.5.2 Add new fridges . . . . .	20
4.5.3 Delete existing fridge . . . . .	21
4.5.4 Detail view of our fridge and the possibility to add and remove food . . . . .	22
4.5.5 add property expirydate to food . . . . .	24
4.6 . . . . .	25

---

<b>II</b>	<b>In-Depth Manual</b>	<b>26</b>
<b>5</b>	<b>Introduction</b>	<b>27</b>
5.1	A new way to write extensions . . . . .	27
5.2	Important terms . . . . .	27
<b>6</b>	<b>New extension structure</b>	<b>28</b>
6.1	Directory structure . . . . .	28
6.2	Class naming conventions . . . . .	28
6.2.1	Interfaces . . . . .	29
6.2.2	Abstract classes . . . . .	29
<b>7</b>	<b>Extbase</b>	<b>30</b>
7.1	Core concepts (Sebastian) . . . . .	30
7.1.1	Model View Controller (MVC) . . . . .	30
7.1.2	Domain Driven Design (DDD) . . . . .	30
7.2	Storing data with domain models (Jochen) . . . . .	31
7.3	Controller (Jochen / Sebastian) . . . . .	31
7.3.1	Actions . . . . .	31
7.3.2	Rendering a response . . . . .	31
7.3.3	Arguments and validation . . . . .	32
7.3.3.1	Advanced validation . . . . .	32
7.4	View (Sebastian) . . . . .	32
7.5	FAQ / Best practices . . . . .	32
<b>8</b>	<b>Fluid (Sebastian)</b>	<b>33</b>
8.1	Basic concepts . . . . .	33
8.1.1	Variables and Object Accessors . . . . .	33
8.1.2	View Helpers . . . . .	34
8.1.2.1	Boolean expressions . . . . .	35
8.1.2.1.1	Simple boolean evaluations . . . . .	35
8.1.2.1.2	Complex boolean expressions . . . . .	35
8.1.3	Arrays . . . . .	36
8.2	Passing data to the view . . . . .	37
8.3	Writing your own View Helper . . . . .	37
8.3.1	Rendering the View Helper . . . . .	37
8.3.2	Declaring arguments . . . . .	38
8.3.2.1	render() method parameters are ViewHelper arguments . . . . .	39
8.3.2.2	initializeArguments() for argument initialization . . . . .	39
8.3.3	TagBasedViewHelper . . . . .	39
8.3.3.1	additionalAttributes . . . . .	41
8.3.4	Facets . . . . .	41
8.3.4.1	SubNodeAccess Facet . . . . .	41
8.3.4.2	PostParse Facet . . . . .	42

---

---

<b>III Reference</b>	<b>43</b>
<b>9 Introduction</b>	<b>44</b>
9.1 A new way to write extensions . . . . .	44
9.2 Important terms . . . . .	44
<b>10 Extbase (Jochen)</b>	<b>45</b>
10.1 Caching . . . . .	45
<b>11 Fluid (Sebastian)</b>	<b>46</b>
11.1 Standard View Helper Library . . . . .	46
11.1.1 alias . . . . .	46
11.1.1.1 Examples . . . . .	46
11.1.1.2 Arguments . . . . .	46
11.1.2 base . . . . .	46
11.1.2.1 Examples . . . . .	47
11.1.2.2 Arguments . . . . .	47
11.1.3 cObject . . . . .	47
11.1.3.1 Arguments . . . . .	47
11.1.4 debug . . . . .	47
11.1.4.1 Arguments . . . . .	47
11.1.5 else . . . . .	47
11.1.5.1 Arguments . . . . .	48
11.1.6 for . . . . .	48
11.1.6.1 Examples . . . . .	48
11.1.6.2 Arguments . . . . .	48
11.1.7 form . . . . .	48
11.1.7.1 Basic usage . . . . .	49
11.1.7.2 A complex form with a specified encoding type . . . . .	49
11.1.7.3 A Form which should render a domain object . . . . .	49
11.1.7.4 Arguments . . . . .	49
11.1.8 form.hidden . . . . .	49
11.1.8.1 Examples . . . . .	49
11.1.8.2 Arguments . . . . .	49
11.1.9 form.select . . . . .	49
11.1.9.1 Basic usage . . . . .	52
11.1.9.2 Pre-select a value . . . . .	52
11.1.9.3 Usage on domain objects . . . . .	52
11.1.9.4 Arguments . . . . .	52
11.1.10 form.submit . . . . .	52

---

---

11.1.10.1 Examples . . . . .	54
11.1.10.2 Arguments . . . . .	54
11.1.11 form.textarea . . . . .	54
11.1.11.1 Examples . . . . .	55
11.1.11.2 Arguments . . . . .	55
11.1.12 form.textbox . . . . .	55
11.1.12.1 Examples . . . . .	56
11.1.12.2 Arguments . . . . .	56
11.1.13 form.upload . . . . .	56
11.1.13.1 Examples . . . . .	57
11.1.13.2 Arguments . . . . .	57
11.1.14 format.crop . . . . .	57
11.1.14.1 Examples . . . . .	58
11.1.14.2 Arguments . . . . .	58
11.1.15 format.currency . . . . .	58
11.1.15.1 Examples . . . . .	59
11.1.15.2 Arguments . . . . .	59
11.1.16 format.date . . . . .	59
11.1.16.1 Examples . . . . .	59
11.1.16.2 Arguments . . . . .	60
11.1.17 format.html . . . . .	60
11.1.17.1 Arguments . . . . .	60
11.1.18 format.nl2br . . . . .	60
11.1.18.1 Arguments . . . . .	61
11.1.19 format.number . . . . .	61
11.1.19.1 Arguments . . . . .	61
11.1.20 format.printf . . . . .	61
11.1.20.1 Examples . . . . .	61
11.1.20.2 Arguments . . . . .	62
11.1.21 if . . . . .	62
11.1.21.1 Arguments . . . . .	62
11.1.22 image . . . . .	62
11.1.22.1 Arguments . . . . .	62
11.1.23 link.action . . . . .	62
11.1.23.1 Examples . . . . .	62
11.1.23.2 Arguments . . . . .	65
11.1.24 link.email . . . . .	65
11.1.24.1 Arguments . . . . .	65
11.1.25 link.external . . . . .	65

---

---

11.1.25.1 Examples . . . . .	65
11.1.25.2 Arguments . . . . .	65
11.1.26 link.page . . . . .	65
11.1.26.1 Examples . . . . .	65
11.1.26.2 Arguments . . . . .	67
11.1.27 then . . . . .	67
11.1.27.1 Arguments . . . . .	67
11.1.28 translate . . . . .	67
11.1.28.1 Arguments . . . . .	67
11.1.29 uri.action . . . . .	67
11.1.29.1 Examples . . . . .	67
11.1.29.2 Arguments . . . . .	70
11.1.30 uri.email . . . . .	70
11.1.30.1 Arguments . . . . .	70
11.1.31 uri.external . . . . .	70
11.1.31.1 Examples . . . . .	70
11.1.31.2 Arguments . . . . .	70
11.1.32 uri.page . . . . .	70
11.1.32.1 Examples . . . . .	71
11.1.32.2 Arguments . . . . .	71
<b>IV Framework reference</b>	<b>72</b>
<b>12 Introduction</b>	<b>73</b>
12.1 A new way to write extensions . . . . .	73
12.2 Important terms . . . . .	73

---

---

## List of Tables

11.1 Arguments . . . . .	47
11.2 Arguments . . . . .	47
11.3 Arguments . . . . .	47
11.4 Arguments . . . . .	48
11.5 Arguments . . . . .	50
11.6 Arguments . . . . .	51
11.7 Arguments . . . . .	53
11.8 Arguments . . . . .	54
11.9 Arguments . . . . .	55
11.10 Arguments . . . . .	56
11.11 Arguments . . . . .	57
11.12 Arguments . . . . .	58
11.13 Arguments . . . . .	59
11.14 Arguments . . . . .	60
11.15 Arguments . . . . .	61
11.16 Arguments . . . . .	61
11.17 Arguments . . . . .	62
11.18 Arguments . . . . .	62
11.19 Arguments . . . . .	63
11.20 Arguments . . . . .	64
11.21 Arguments . . . . .	65
11.22 Arguments . . . . .	66
11.23 Arguments . . . . .	68
11.24 Arguments . . . . .	69
11.25 Arguments . . . . .	69
11.26 Arguments . . . . .	70
11.27 Arguments . . . . .	70
11.28 Arguments . . . . .	71

---

## **Abstract**

Extbase is the next-generation framework for building TYPO3 v4 extensions

## **Part I**

**Quickstart, get `blog_example` running and try  
your own first steps (Franz Ripfel)**

---

# Chapter 1

## Introduction

This book is part of a bigger document about *a new way to write extensions*.

The whole document is structured into the following books:

- *Quickstart*: a hands-on example on how to write extensions with Extbase and Fluid
- *In-depth manual*: explaining all the details you need to know to use the system
- *Reference*: Containing all reference materials, conventions, ...
- *Developer reference*: Showing the framework internals for people wanting to get into framework programming

All books share the same introduction, so if this chapter looks familiar to you, just skip it!

### 1.1 A new way to write extensions

Extensions were introduced in TYPO3 3.5, and are one of the main reasons TYPO3 is so flexible, popular and widely used. However, the base class used for frontend extensions (called `tslib_pibase` - hence, we call old style extensions *pibase-style extensions*) has not changed a lot since its introduction a few years ago.

However, in the meantime many more advanced development concepts came into more widespread use in the PHP and TYPO3 community, especially the Model-View-Controller design pattern. We strongly feel the benefits of using such modern paradigms, as they provide more structure, guidance and help to the extension programmer and facilitate understanding foreign extensions.

All of the concepts introduced in this manual have been implemented by the TYPO3 v5 / FLOW3 team, and Extbase is a backport of the MVC and DDD functionality of FLOW3. We'd like to thank the many people involved in developing FLOW3 or helped with backporting the code to TYPO3 v4. Your input made this project a true community project.

### 1.2 Important terms

Here, we want to give some brief definitions of the most needed terms we'll use throughout the documents.

- **Model View Controller (MVC)**: A widely used design pattern which identifies three major concerns in every software project: A data model, a View to display the data to the user, and a Controller to handle user interaction and data flow.
  - **Domain Driven Design (DDD)**: A concept of structuring the Model. Its idea is to model the real world with objects, and implement both their behavior and data. It emphasizes the separation of the model and the repositories to access the model.
  - **FLOW3**: The next-generation enterprise PHP framework used as a basis to TYPO3 5.0
  - **Extbase**: A backport of the MVC and DDD functionality of FLOW3 to TYPO3 v4
  - **Fluid**: A templating engine designed for ease of use, flexibility and extensibility. It was specifically developed for FLOW3, and has been backported to TYPO3 v4 as well
-

## Chapter 2

# Get the official demo `blog_example` running

Here you will install the `blog_example` and see it running.

### 2.1 Prerequisites: TYPO3 4.3 with pagetree and working basic layout

Extbase requires TYPO3 4.3. Download it from [www.typo3.org](http://www.typo3.org) and install it like your other TYPO3 installations before.

First of all you will need at least a minimalistic pagetree to be able to place the `blog_example` plugin onto some page. Create a rootpage (and some subpages if you want). We suggest you to have a startpage (with the TypoScript-Template in it) and some subpages for testing the different available plugins, for now this would be on page for the `blogexample` and one for the fluid tester (EXT: `viewhelpertest`).

In order to see some output from our `blog_example` you need to have some general layout already working. The autor's personal favorit layout for just playing around with plugins is the static template "GLUECK". You should know how to get a basic template running, otherwise you are probably not yet ready to play around with Extbase. Check out <http://typo3.org/documentation/> for more details in case you are in doubt about how to install TYPO3 and get get a basic system running.

### 2.2 Install extensions from MVC-Project and add plugins to page

Install the following extensions and follow the instructions during installation (just like with other extensions): **extbase**, **fluid**, **viewhelpertest**, **blog\_example**.

To get some output in the frontend, insert the plugin 'A blog example' and set the startingpoint to the same page where you inserted the plugin. If you call the page "blogexample" you should see the `blog_example` welcome page.

---

**Note**

You can add the Plugin 'Fluid Tester' to another test page (see screenshot above: fluid tester) to see some basic fluid template options in action and get a feeling how it works. For sure later on we will inspect also the html templates done with fluid in `blog_example`.

---

Hurry, now you should see some output of `blog_example`, your first working Extbase extension!

### 2.3 Check out the functionality of `blog_example`

Now that the `blog_example` basically seems to work, you should play around a little, try adding some demo data or add you own blog, see or add blogposts, see or add comments and so on. It's just what a simple blog does.

---

---

**Note**

At the time of writing this quickstart there was still an issue about caching to be solved: If you are not logged in to the backend for testing purposes it is best to set `config.no_cache=1` for now.

---

## Chapter 3

# Understand the important parts and see how it works

Now that the demo is successfully running and it proved to be running correctly, you surely want to get some insights.

### 3.1 Filestructure and naming conventions

The default directory structure of Extbase based extensions is meant to resemble FLOW3 packages. Thus, the naming conventions shown here are the same as in FLOW3. For a detailed overview about the filestructure see chapter "New extension structure". For now, you just have to inspect `blog_example`, the filestructure there is pretty straightforward to understand.

---

**Tip**

Mind the class names, which are always based on the file structure: `Tx_BlogExample_Controller_BlogController -> blog_example/Classes/Controller/BlogController.php`

---

### 3.2 How TYPO3 is notified to insert our plugin

The magic is done by calling `Tx_Extbase_Utility_Plugin::registerPlugin` in `ext_tables.php`. Basically it is an extended version of the old `t3lib_extMgm::addPlugin`. For a deeper insight inspect the method `Tx_Extbase_Utility_Plugin::registerPlugin`. By now you should know how to find the class-file just by reading the class name, shouldn't you?

Especially important is to think of which parts should be cached (USER-Object) and which parts should not be cached (USER\_INT-Object). You know this decision from traditional frontend plugin. The new and cool think is that with Extbase you can define very finely which combination of controller and action should be cached. As a rule of thumb you could say: All actions modifying data in the repository (which is your access point to the database) should result in non-cached objects.

---

**Note**

At the time of writing this quickstart there was still an issue about caching to be solved: Already cached views, which are redirected to from uncached views still show the cached version even if data has changed. We need to find a smart way to flush depending cache entries. <http://forge.typo3.org/issues/show/3421>

---

With Extbase TYPO3 will call the same dispatcher for all Extbase plugins (`userFunc = tx_extbase_dispatcher->dispatch`, have a look at `Tx_Extbase_Utility_Plugin::registerPlugin`), this is different to the traditional way, where by default the main-function of the individual plugin was called. Therefore the dispatcher is your main starting point for finding out what's going on in the Extbase world.

Mind the comment "`// The first controller and its first action will be the default`". We repeat: TYPO3/Extbase will call the first controller in that array and then the first given action in the action list. Mind that for your own extensions.

---

### 3.3 Inspect the blog controller

Let's have a look into the class `Tx_BlogExample_Controller_BlogController`. You can find a bunch of actions in there. In `ext_tables.php` we defined that `indexAction()` is called by default if no other action is requested. Since the controller has to decide what to display and how to get the correct data for the display, often you see there some code related to view-classes and to repository-classes.

---

**Example 3.1** `Tx_BlogExample_Controller_BlogController::indexAction()`

---

```
/**
 * Index action for this controller. Displays a list of blogs.
 *
 * @return string The rendered view
 */
public function indexAction() {
    $this->view->assign('blogs', $this->blogRepository->findAll());
}
```

This single line of code triggers quite a lot of magic. It gets all blogs from the `blogRepository` as array of `Blog`-Objects and assigns this to the variable 'blogs' within the fluid template. There we can just iterate through this Objects and retrieve the needed elements for displaying them in the frontend. We will have a closer look into the html-templates in the next section.

---

---

**Example 3.2** `Tx_BlogExample_Controller_BlogController::updateAction()`

---

```
/**
 * Updates an existing blog
 *
 * @param Tx_BlogExample_Domain_Model_Blog $blog The existing, unmodified blog
 * @param Tx_BlogExample_Domain_Model_Blog $updatedBlog A clone of the original blog with ←
 *         the updated values already applied
 * @return void
 */
public function updateAction(Tx_BlogExample_Domain_Model_Blog $blog, ←
    Tx_BlogExample_Domain_Model_Blog $updatedBlog) {
    $this->blogRepository->replace($blog, $updatedBlog);
    $this->redirect('index');
}
```

The `updateAction` is called if you save the blog edit form in the frontend. With this two lines of code we tell the `blogRepository` that it should replace the current blog data with our (edited) new blog data and then just hand over to `indexAction()`, which in turn will show all blogs. Now what is especially interesting and probably the most astonishing part of it is the automatic creation of a blog object out of form data because of the definitions in the PHPDoc `@param` above.

Here is not the place to explain that in detail, just keep in mind to always check the PHPDoc to be sure to provide the correct classnames thus the framework can provide correct objects as params to the method calls. We will talk more about that later.

---

### 3.4 Inspect the html-templates (and the folder structure they are in)

Did you already realise that our folder `blog_example/Classes/View` is empty? Hey, shouldn't we have a view to get some output to the frontend? The answer is yes, we should and that needed basic view is already delivered within Extbase. If we do not have some special needs for our output we just have to place our html-templates into the right folder and there they are found automatically by Extbase. Cool, eh? So let's have a look into `Resources/Private/Templates/Blog`. What would you think is the html-template used for the blogexample start page, which originates in `indexAction()`? Right, it's `index.html`! Let's have a look into it.

---

**Example 3.3** index.html

```

<div class="csc-header csc-header-n1"><h1 class="csc-firstHeader">Welcome to the Blog ←
  Example</h1></div>
<p class="bodytext">This installation serves as a demonstration for basic development ←
  techniques used in Extbase/FLOW3 applications. Please note that this example might be ←
  not fully functional at certain times because we're continuously working on it and it ←
  has not yet been officially released.</p>
<p class="bodytext">Here is a list of blogs:</p>
<ul>
  <f:for each="{blogs}" as="blog">
    <li>
      <h3><f:link.action controller="Post" action="index" arguments="{blog : blog}">{blog. ←
        name}</f:link.action></h3>
      <p>
        <f:format.Nl2br>{blog.description}</f:format.Nl2br><br />
        <f:link.action action="show" arguments="{blog : blog}">Show</f:link.action>
        <f:link.action action="edit" arguments="{blog : blog}">Edit</f:link.action>
        <f:link.action action="delete" arguments="{blog : blog}">Delete</f:link.action>
      </p>
    </li>
  </f:for>
</ul>
<p><f:link.action action="new">Create a blog</f:link.action></p>
<p><f:link.action action="populate">Create example data</f:link.action></p>
<p><f:link.action action="deleteAll">Delete all Blogs [!!!]</f:link.action></p>

```

Try to edit the text in it, clear the frontend cache and see if your changes are reflected in the frontend. If nothing changes, you are either in the wrong html-template or on the wrong page in the frontend. From this point on you should be able to figure out which html-template is used in which part of our blogexample. Do you see the connection between the folder names and the names of the controller files and classnames?

Blog -> BlogController, Post -> PostController

indexAction -> index.html, newAction -> new.html

This is not just coincidence, this is the way it has to be to work properly without special configuration!

Watch out for the part {blogs} in the for-each-construct. Do you remember? In the BlogController we assigned all the blogs to exactly this variable and are now able to get the required information in our html-template. See next section for a better understanding of how to use Fluid, the new templating engine.

### 3.5 Fluid: Get a basic understanding how the html-templates work

In this chapter you will just get a first glimpse of how to use Fluid, this is not the place to get an in-depth-view of fluid. Because our TYPO3v4 version of Fluid is a backport from the FLOW3 effort, the latest FLOW3 release is the place to go for the currently most detailed documentation available. See Subversion path <https://svn.typo3.org/FLOW3/Packages/Fluid/trunk/Documentation/Manual/Doc> (that path might change because of ongoing development, but you get a good hint where to search for it).

In TYPO3v4, which is always compatible with PHP 5.2, it is not possible to use Namespaces, which are introduced in PHP 5.3. That's why we omit the namespace declarations in Fluid. The syntax should be quite the same. Just compare our blog\_example html templates with the ones used in FLOW3. If you see a tag like <f3:xxx> then you just found an old declaration. That base tag was renamed from f3 to f.

All the work is done by so-called view helpers. For each view helper, there exists a corresponding PHP class. You find the basic ones in fluid/Classes/ViewHelpers/. Remember, the naming conventions are helping you to find the correct class if you are coming from the template or - the other way round - knowing how to write the correct xml-tag based on classnames. To check out the code for the <f:for>-tag, you have to have a look into fluid/Classes/ViewHelpers/ForViewHelper.php.

Check out the extension viewhelpertest mentioned in Chapter 2 with its plugin to see some examples of using the available view helpers.

## 3.6 Persistence: Or where are the database calls?

You are probably used to retrieve data from the database and write data to the database via your own SQL-Statements. With Extbase there will be a new layer between your business logic and the database: The repository.

Let's have a look into the folder `blog_example/Classes/Domain/Model/`. You see files representing your model objects `Blog`, `Post`, `Comment` and `Tag`. Beside these files you see a file/class called `BlogRepository` and `PostRepository`. Both these classes found in these files extend `Tx_Extbase_Persistence_Repository` and are your "contact persons" to retrieve and save data. This means that you - at least in regular circumstances - do not need to write SQL-queries anymore, but call the appropriate methods of your repository. This will help you to concentrate on the business logic. Because the execution of every regular request comes through the dispatcher (class `Tx_Extbase_Dispatcher`) this is the perfect place to handle the persistence.

---

### Tip

Since Extbase is still under development, the already out-of-the-box-supported database relations are limited. Check the newsgroup `typo3.projects.typo3v4mvc` on `news.typo3.org` if you think you need a so-far unsupported relation type.

---

### 3.6.1 Retrieve specific data (show the blog edit form with the current data of the blog)

If you have already inspected the `BlogController` (i.e. in the method `indexAction`) as described above you've probably noticed that with a call like `$this->blogRepository->findAll()` you can retrieve ready-made objects. Have a look at `Tx_Extbase_Persistence_Repository` to see what basic methods there are to retrieve data from your Repositories. What is more difficult to understand is what happens if you want to edit an already created blog:

The Controller again needs quite little code:

---

#### Example 3.4 `Tx_BlogExample_Controller_BlogController::editAction()`

```
/**
 * Edits an existing blog
 *
 * @param Tx_BlogExample_Domain_Model_Blog $blog The original blog
 * @return string Form for editing the existing blog
 */
public function editAction(Tx_BlogExample_Domain_Model_Blog $blog) {
    $this->view->assign('blog', $blog);
}
```

---

Hum, do you see any code which might indicate that we are retrieving data from the database? Well, here is some special magic at work. The `editAction()` is called because we clicked the edit link (ie. `index.php?id=3&tx_blogexample_pi1[blog][uid]=17&tx_blogexample_pi1[blog][title]=My new blog`) thus send the information that we want to edit the blog with uid 17. Since Extbase wants to work with objects, it automatically instantiates a `Blog` Object and fills it with the corresponding data. For now you just have to know that you have to add the comment `@param [ClassName_for_object_you_want_to_have_created_from_your_given_id]` to inform Extbase what an object you need. If you want to dig deeper into what's going on, have a look into the `Tx_Extbase_MVC_Controller_ActionController->mapRequestArgumentsToControllerArguments()`.

### 3.6.2 Save / persist data (save/persist the modified data from the form)

The Controller again needs quite little code:

---

---

**Example 3.5** Tx\_BlogExample\_Controller\_BlogController::updateAction()

```
/**
 * Updates an existing blog
 *
 * @param Tx_BlogExample_Domain_Model_Blog $blog The existing, unmodified blog
 * @param Tx_BlogExample_Domain_Model_Blog $updatedBlog A clone of the original blog with ←
 *         the updated values already applied
 * @return void
 */
public function updateAction(Tx_BlogExample_Domain_Model_Blog $blog, ←
    Tx_BlogExample_Domain_Model_Blog $updatedBlog) {
    $this->blogRepository->replace($blog, $updatedBlog);
    $this->redirect('index');
}
```

---

You just have to tell your BlogRepository to replace the existing blog with your updatedBlog, the rest is done by the framework. In order to get the parameter \$updatedBlog properly you need to name the edit form "updateBlog". Check out the file Resources/Private/Templates/Blog/edit.html, there you will find a line like <f:form method="post" controller="Blog" action="update" name="**updatedBlog**" object="{blog}" arguments="{blog: blog}" >

With creating a new Blog entry it is quite similar. Have a look in Tx\_BlogExample\_Controller\_BlogController::createAction() and Resources/Private/Templates/Blog/new.html

## Chapter 4

# Your Turn: Building you own first Extbase-extension

---

### Note

This part is not finished yet, but it might be helpful already. Please be patient, feedback (to the Extbase mailing list) is welcome nevertheless. Due to ongoing changes and new development there might even be some errors in here. If you use the new kickstarter to create your extension, there are probably some differences in the created code and the code snippets you will see in this manual. We will adapt the code here once the kickstarter has reached its final stage.

---

Now let's get started: its your turn to build your own first (and simple) Extbase based extension! It's always good to see some examples, so it might be a good idea to check the TER or forge for new extbase based extensions. Just search for Extbase and you should get also the Extbase based extensions. One very simple starting example ist efempty from Patrick Lobacher. Thanks Patrick! There is no functionality, but you get the basic structure on which you could build some first tests.

## 4.1 Discuss and decide a Specification

The first step for your successful extension based on Extbase (and we are sure you did the same for your old-style extensions) is to get a clear picture of what should be done by the extension, generally known as feature list.

Let's assume your partner always complains that things get rotten in your frigde. Now since your are a technically skilled person, you decide to solve this with the force of IT and create a little web application to help you with controlling the status of your fridges.

The needed feature list for your first alpha release consists of of the following elements:

- you shold be able to add, remove and edit fridges and identify them by their location. You probably have on in the kitchen and one in the cellar, haven't you?
- you want to be able to add and remove items for each fridge, defining the items by name and date of expiry.
- for both features above you want to have a nice list view.

## 4.2 Design your model

---

---

## 4.3 Build your model (part1)

---

### Tip

As soon as the Extbase kickstarter is finished you will just have to design your model within the kickstarter and it will take care of creating the needed model objects and repositories. Check out the current status on forge, this looks very promising, thanks to Ingmar. (Search for `extbase_kickstarter`)

---

Alternatively you will have to write all the needed classes for yourself. Don't worry, this will help you to understand the paradigms and naming conventions better :)

Without the new kickstarter you have to care yourself for extracting the database model from the Domain Model, here are your thoughts: You need two new tables, fridge and food. Because we have to care about naming conventions they will have the names `tx_frigdemaster_domain_model_fridge` and `tx_frigdemaster_domain_model_food`. For the fields, see the screenshot of the kickstarter above.

The resulting files should be as following:

---

**Example 4.1** ext\_tables.php

```

<?php
if (!defined ('TYPO3_MODE')) {
    die ('Access denied.');
```

---

```

}

/**
 * A fully configured omnipotent plugin
 */
Tx_Extbase_Utility_Extension::registerPlugin(
    $_EXTKEY, // The name of the extension in UpperCamelCase
    'Pil', // A unique name of the plugin in ←
    UpperCamelCase
    'My Fridgemaster' // A title shown in the backend dropdown ←
    field
);

t3lib_extMgm::allowTableOnStandardPages('tx_fridgemaster_domain_model_fridge');

$TCA['tx_fridgemaster_domain_model_fridge'] = array (
    'ctrl' => array (
        'title' => 'LLL:EXT:fridgemaster/Resources/Private/Language/locallang_db.xml: ←
            tx_fridgemaster_domain_model_fridge',
        'label' => 'location',
        'tstamp' => 'tstamp',
        'crdate' => 'crdate',
        'cruser_id' => 'cruser_id',
        'default_sortby' => 'ORDER BY crdate',
        'delete' => 'deleted',
        'enablecolumns' => array (
            'disabled' => 'hidden',
        ),
        'dynamicConfigFile' => t3lib_extMgm::extPath($_EXTKEY).'Configuration/TCA/tca.php',
        'iconfile' => t3lib_extMgm::extRelPath($_EXTKEY).'Resources/Public/Icons/ ←
            icon_tx_fridgemaster_domain_model_fridge.gif',
    ),
);

t3lib_extMgm::allowTableOnStandardPages('tx_fridgemaster_domain_model_food');

$TCA['tx_fridgemaster_domain_model_food'] = array (
    'ctrl' => array (
        'title' => 'LLL:EXT:fridgemaster/Resources/Private/Language/locallang_db.xml: ←
            tx_fridgemaster_domain_model_food',
        'label' => 'name',
        'tstamp' => 'tstamp',
        'crdate' => 'crdate',
        'cruser_id' => 'cruser_id',
        'default_sortby' => 'ORDER BY crdate',
        'delete' => 'deleted',
        'enablecolumns' => array (
            'disabled' => 'hidden',
        ),
        'dynamicConfigFile' => t3lib_extMgm::extPath($_EXTKEY).'Configuration/TCA/tca.php',
        'iconfile' => t3lib_extMgm::extRelPath($_EXTKEY).'Resources/Public/Icons/ ←
            icon_tx_fridgemaster_domain_model_food.gif',
    ),
);

t3lib_div::loadTCA('tt_content');
$TCA['tt_content']['types']['list']['subtypes_excludelist'][$_EXTKEY.'_pil']='layout, ←
    select_key';
?>
```

---

**Example 4.2** ext\_localconf.php

---

```
<?php
if (!defined ('TYPO3_MODE')) die ('Access denied.');
```

```
/**
 * Configure the Plugin to call the
 * right combination of Controller and Action according to
 * the user input (default settings, FlexForm, URL etc.)
 */
Tx_Extbase_Utility_Extension::configurePlugin(
    $_EXTKEY, // The extension name (in UpperCamelCase) or ↵
    the extension key (in lower_underscore)
    'Pil', // A unique name of the plugin in ↵
    UpperCamelCase
    array( // An array holding the controller-action- ↵
        combinations that are accessible
        'Fridge' => 'index,show,new,create,delete,edit,update', // The first controller and its ↵
            first action will be the default
        'Food' => 'add,remove',
    ),
    array( // An array of non-cachable controller-action ↵
        -combinations (they must already be enabled)
        'Fridge' => 'index,show,new,create,delete,edit,update', // The first controller and its ↵
            first action will be the default
        'Food' => 'add,remove',
    )
);
?>
```

---

For a beginning we mark all combinations as none cachable to avoid irritations because of caching issues.

---

**Example 4.3** ext\_tables.sql

---

```
#
# Table structure for table 'tx_fridgemaster_domain_model_fridge'
#
CREATE TABLE tx_fridgemaster_domain_model_fridge (
  uid int(11) NOT NULL auto_increment,
  pid int(11) DEFAULT '0' NOT NULL,
  tstamp int(11) DEFAULT '0' NOT NULL,
  crdate int(11) DEFAULT '0' NOT NULL,
  cruser_id int(11) DEFAULT '0' NOT NULL,
  deleted tinyint(4) DEFAULT '0' NOT NULL,
  hidden tinyint(4) DEFAULT '0' NOT NULL,
  location tinytext,
  foods int(11) DEFAULT '0' NOT NULL,

  PRIMARY KEY (uid),
  KEY parent (pid)
);

#
# Table structure for table 'tx_fridgemaster_domain_model_food'
#
CREATE TABLE tx_fridgemaster_domain_model_food (
  uid int(11) NOT NULL auto_increment,
  pid int(11) DEFAULT '0' NOT NULL,
  tstamp int(11) DEFAULT '0' NOT NULL,
  crdate int(11) DEFAULT '0' NOT NULL,
  cruser_id int(11) DEFAULT '0' NOT NULL,
  deleted tinyint(4) DEFAULT '0' NOT NULL,
  hidden tinyint(4) DEFAULT '0' NOT NULL,
  name tinytext,
  expirydate int(11) DEFAULT '0' NOT NULL,
  fridge int(11) DEFAULT '0' NOT NULL,
  fridge_table tinytext NOT NULL,

  PRIMARY KEY (uid),
  KEY parent (pid)
);
```

---

**Example 4.4** Configuration/Tca/tca.php

```

<?php
if (!defined ('TYPO3_MODE')) die ('Access denied.');
```

---

```

$TCA['tx_fridgemaster_domain_model_fridge'] = array (
    'ctrl' => $TCA['tx_fridgemaster_domain_model_fridge']['ctrl'],
    'interface' => array (
        'showRecordFieldList' => 'hidden,location'
    ),
    'feInterface' => $TCA['tx_fridgemaster_domain_model_fridge']['feInterface'],
    'columns' => array (
        'hidden' => array (
            'exclude' => 1,
            'label' => 'LLL:EXT:lang/locallang_general.xml:LGL.hidden',
            'config' => array (
                'type' => 'check',
                'default' => '0'
            )
        ),
        'location' => array (
            'exclude' => 0,
            'label' => 'LLL:EXT:fridgemaster/Resources/Private/Language/locallang_db.xml:↔
                tx_fridgemaster_domain_model_fridge.location',
            'config' => array (
                'type' => 'input',
                'size' => '30',
            )
        ),
        'foods' => array(
            'exclude' => 0,
            'label' => 'LLL:EXT:fridgemaster/Resources/Private/Language/locallang_db.xml:↔
                tx_fridgemaster_domain_model_fridge.foods',
            'config' => array(
                'type' => 'inline',
                'foreign_class' => 'Tx_Fridgemaster_Domain_Model_Food',
                'foreign_table' => 'tx_fridgemaster_domain_model_food',
                'foreign_field' => 'fridge',
                'foreign_table_field' => 'fridge_table',
                'appearance' => array(
                    'newRecordLinkPosition' => 'bottom',
                    'collapseAll' => 1,
                    'expandSingle' => 1,
                )
            ),
        ),
    ),
    'types' => array (
        '0' => array('showitem' => 'hidden;;;1-1-1, location,foods')
    ),
    'palettes' => array (
        '1' => array('showitem' => '')
    )
);

$TCA['tx_fridgemaster_domain_model_food'] = array (
    'ctrl' => $TCA['tx_fridgemaster_domain_model_food']['ctrl'],
    'interface' => array (
        'showRecordFieldList' => 'hidden,name,expirydate,fridge'
    ),
    'feInterface' => $TCA['tx_fridgemaster_domain_model_food']['feInterface'],
    'columns' => array (
        'hidden' => array (
            'exclude' => 1,
            'label' => 'LLL:EXT:lang/locallang_general.xml:LGL.hidden',
            'config' => array (
                'type' => 'check',
                'default' => '0'
            )
        )
    )
);

```

Now if that looks good, install the extension and create a testpage where we will place our frontend plugin later.

just to make sure that your TCA was created correctly, you should add your first fridge and some food directly in the backend to the newly created page. This way, we have already some sample data to show in the frontend later.

Now that we created the basic TCA stuff, let's get our folder structure correctly (or check if the kickstarter did it the right way). You can find some more details in the main manual. Finally it should look like this:

Check for proper places (see `blog_example` for correct example):

- the icons to the folder Icons
- `locallang.php` and `locallang_db.php` in folder Languages, check your `ext_tables.php` to reflect the correct path in all LLL definitions.
- `tca.php` in folder TCA, check `ext_tables.php` to reflect the correct path in all `dynamicConfigFile` declarations.

---

**Note**

As long as there is no final Extbase kickstarter to help with all that filestructure and needed classes we want to check as early as possible if we are doing this correct. Because of that, we start with only a part of our task and try to get an valid output to the frontend (in our example just the list of fridges. All the other functionality and corresponding objects we will add later.

---

- create/check first model object: fridge. Have a look into `blog_example`, the object `Blog` there can be used as a blueprint. Pay special attention to the correct classname: `Tx_Fridgemaster_Domain_Model_Fridge`
-

---

**Example 4.5** Tx\_Fridgemaster\_Domain\_Model\_Fridge

---

```
/**
 * A Fridge
 *
 * @version $Id:$
 * @copyright Copyright belongs to the respective authors
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License, version 2
 * @scope prototype
 * @entity
 */
class Tx_Fridgemaster_Domain_Model_Fridge extends Tx_Extbase_DomainObject_AbstractEntity {

    /**
     * The fridge's location
     *
     * @var string
     */
    protected $location = '';

    /**
     * The food items contained in this fridge
     *
     * @var array
     */
    #protected $foods = array();

    /**
     * Constructs this fridge
     *
     * @return
     */
    public function __construct() {
    }

    /**
     * Sets this fridge's location
     *
     * @param string $location The fridge location
     * @return void
     */
    public function setLocation($location) {
        $this->location = $location;
    }

    /**
     * Returns the fridge's location
     *
     * @return string The fridge's location
     */
    public function getLocation() {
        return $this->location;
    }
}
```

---

**Note**

The property \$foods is deactivated (commented out) on purpose, we want to care about this relations later.

---

- create/check repository FridgeRepository, use BlogRepository as blueprint

---

---

**Example 4.6 Tx\_Fridgemaster\_Domain\_Repository\_FridgeRepository**

---

```
class Tx_Fridgemaster_Domain_Repository_FridgeRepository extends ↵  
    Tx_Extbase_Persistence_Repository {  
}
```

---

You wonder why this class is empty? Well, our currently needed features are perfectly handled by the parent class `Tx_Extbase_Persistence_Repository`, so for now, no extra code needed.

## 4.4 Add Controller / View (part1)

Now that we've implemented the first part of our model, let's see if we can already get some results to the frontend. Insert a plugin and set the startingpoint to the same page where you are creating the plugin.

Are you getting an error about a missing `FridgeController`? Well, the good news is that Extbase is already pointing to `FridgeController`, it's just not yet there. So let's create a `FridgeController`, and if you need some help, just use the `BlogController` again as a blueprint:

---

**Example 4.7 Tx\_Fridgemaster\_Controller\_FridgeController**

---

```
class Tx_Fridgemaster_Controller_FridgeController extends ↵  
    Tx_Extbase_MVC_Controller_ActionController {  
  
    /**  
     * @var Tx_Fridgemaster_Domain_Repository_FridgeRepository  
     */  
    protected $fridgeRepository;  
  
    /**  
     * Initializes the current action  
     *  
     * @return void  
     */  
    public function initializeAction() {  
        $this->fridgeRepository = t3lib_div::makeInstance(' ↵  
            Tx_Fridgemaster_Domain_Repository_FridgeRepository');  
    }  
  
    /**  
     * Index action for this controller. Displays a list of friges.  
     *  
     * @return string The rendered view  
     */  
    public function indexAction() {  
        $this->view->assign('friges', $this->fridgeRepository->findAll());  
    }  
}
```

---

If the error message is gone, but still no content shows up, it's because there is no HTML template so far. By simply adding `index.html` in `Resources/Private/Templates/Fridge/`, or copying the `index.html` from the `blog_example` extension (use `Resources/Private/Templates/Blog/index.html` from `blog_example` and modify it to your variables), you should get the list of fridges in the plugin.

---

---

**Example 4.8** index.html

---

```
<div class="csc-header csc-header-n1"><h1 class="csc-firstHeader">Welcome to the Master of ↵  
  Fridges</h1></div>  
<p class="bodytext">Here are your fridges:</p>  
<ul>  
  <f:for each="{fridges}" as="fridge">  
    <li>  
      <p>  
        {fridge.location}<br />  
        <f:link.action action="edit" arguments="{fridge : fridge}">Edit</f:link.action>  
        <f:link.action action="delete" arguments="{fridge : fridge}">Delete</f:link.action>  
      </p>  
    </li>  
  </f:for>  
</ul>  
<p><f:link.action action="new">Create a new fridge</f:link.action></p>
```

---

Now you should have your first success! You have your default screen with your Backend-created Fridge on the screen. From now on it will be easier, we will just have to adapt our knowledge.

## 4.5 Expand to whole specification

To have a nicely working demo application, we will add additional screens with their appropriate MVC classes.

### 4.5.1 Edit existing fridges

We want to fill the edit-link on our startpage with a real form. For that we need to decide on an appropriate action (editAction) and the corresponding html-template (edit.html). Again we use blog\_example as the blueprint to get an idea of what to do.

---

**Example 4.11** editAction in Tx\_Fridgemaster\_Controller\_FridgeController

```
/**
 * Index action for this controller. Displays a list of friges.
 *
 * @param Tx_Fridgemaster_Domain_Model_Fridge $fridge The original fridge
 * @return string The rendered view
 */
public function editAction(Tx_Fridgemaster_Domain_Model_Fridge $fridge) {
    $this->view->assign('fridge', $fridge);
}
```

The editAction() looks quite similar to indexAction(). Remember what you have learned (or if not, have a another look) from the first part of this quickstart tutorial, when we analysed the editAction in BlogController.

**Example 4.10** edit.html

```
<div class="csc-header csc-header-n1"><h1 class="csc-firstHeader">Edit fridge from {fridge. ↵
    location}</h1></div>
<p class="bodytext">Edit the information about your fridge below:</p>
<f:form method="post" controller="Fridge" action="update" name="updatedFridge" object="{ ↵
    fridge}" arguments="{fridge: fridge}" >
    <label for="location">Location</label>
    <f:form.textbox property="location" /><br /><br />
    <f:form.submit>Send</f:form.submit>
</f:form>
```

Because we used the blog\_example edit.html as blueprint we noticed that we should define an action already as target for the saving of the edited data. Again, remember what you've learned with the blog\_example about the attributes of the f:form element. When prepared like in the above edit.html, we just need to implement the updateAction to be able to save our own first edited data.

**Example 4.9** updateAction in Tx\_Fridgemaster\_Controller\_FridgeController

```
/**
 * Updates an existing fridge
 *
 * @param Tx_Fridgemaster_Domain_Model_Fridge $fridge The existing, unmodified fridge
 * @param Tx_Fridgemaster_Domain_Model_Fridge $updatedFridge A clone of the original fridge ↵
    with the updated values already applied
 * @return void
 */
public function updateAction(Tx_Fridgemaster_Domain_Model_Fridge $fridge, ↵
    Tx_Fridgemaster_Domain_Model_Fridge $updatedFridge) {
    $this->fridgeRepository->replace($fridge, $updatedFridge);
    $this->redirect('index');
}
```

**4.5.2 Add new fridges**

To be able to add a new Fridge we need an action which will show the form for entering the data for the new fridge. This action will be called from our new-link. We already prepared that link in our index.html above.

```
<f:link.action action="new">Create a new fridge</f:link.action>
```

**Example 4.12** newAction in Tx\_Fridgemaster\_Controller\_FridgeController

```
/**
 * new action for this controller. Displays a form for a new fridge.
 *
 * @param Tx_Fridgemaster_Domain_Model_Fridge $newFridge A fresh Fridge object not yet in ↵
 *   the repository
 * @return string The rendered view
 */
public function newAction(Tx_Fridgemaster_Domain_Model_Fridge $newFridge=null) {
    $this->view->assign('newFridge', $newFridge);
}
```

Within the form we need to define which action will receive our data and handle it. Furthermore we have to make sure to send the data with the correct naming. The naming here must match the naming in the createAction! Have a look into the FormViewHelper (in EXT: fluid) to get more details

**Example 4.13** new.html

```
<div class="csc-header csc-header-n1"><h1 class="csc-firstHeader">New fridge </h1></div>
<p class="bodytext">Edit the information about your fridge below:</p>
<f:form method="post" controller="Fridge" action="create" name="newFridge" object="{ ↵
    newFridge}">
    <label for="location">Location</label>
    <f:form.textbox property="location" /><br /><br />
    <f:form.submit>Send</f:form.submit>
</f:form>
```

The createAction then has no more to do than to inform the fridgeRepository that it should add this new fridge and redirect to the list, which should now show already the new fridge.

**Example 4.14** createAction in Tx\_Fridgemaster\_Controller\_FridgeController

```
/**
 * Creates a new fridge
 *
 * @param Tx_Fridgemaster_Domain_Model_Fridge $newFridge A fresh Fridge object which has ↵
 *   not yet been added to the repository
 * @return void
 */
public function createAction(Tx_Fridgemaster_Domain_Model_Fridge $newFridge) {
    $this->fridgeRepository->add($newFridge);
    $this->redirect('index');
}
```

**4.5.3 Delete existing fridge**

As straightforward as it could be! We already prepared the delete link in our index.html above.

```
<f:link.action action="delete" arguments="{fridge : fridge}">Delete</f:link.action>
```

**Example 4.15** deleteAction in Tx\_Fridgemaster\_Controller\_FridgeController

```

/**
 * Deletes an existing Fridge
 *
 * @param Tx_Fridgemaster_Domain_Model_Fridge $fridge The fridge, which is to be deleted
 * @return void
 */
public function deleteAction(Tx_Fridgemaster_Domain_Model_Fridge $fridge) {
    $this->fridgeRepository->remove($fridge);
    $this->redirect('index');
}

```

**4.5.4 Detail view of our fridge and the possibility to add and remove food**

todo: unchecked for changes in Extbase from here on, probably errors!

This time we can use the editAction as blueprint. Since with the editAction we also show data of a single fridge, it has to be quite similar. We want to link the location of our fridges to get a detailed (single) view. Therefore we need to add that link to our index.html.

{fridge.location} will be replaced with <f:link.action action="show" arguments="{fridge : fridge}">{fridge.location}</f:link.action>

**Example 4.16** showAction in Tx\_Fridgemaster\_Controller\_FridgeController

```

/**
 * Show action for this controller. Displays a single fridge.
 *
 * @param Tx_Fridgemaster_Domain_Model_Fridge $fridge The choosen fridge
 * @return string The rendered view
 */
public function showAction(Tx_Fridgemaster_Domain_Model_Fridge $fridge) {
    $this->view->assign('fridge', $fridge);
}

```

As usual there will be the template show.html to match to our new showAction. In there we want to have a list of all food items added to the fridge.

**Example 4.17** show.html

```

<div class="csc-header csc-header-n1"><h1 class="csc-firstHeader">fridge: {fridge.location} </h1></div>
<ul>
  <f:for each="{fridge.foods}" as="food">
    <li>
      <p>
        {food.name}
      </p>
    </li>
  </f:for>
</ul>

```

At this point you probably will see no food items, even if you already added some food into a fridge in the backend. That's because we didn't care so far for the food. We deactivated the property foods in the Fridge object (see codeexample of class Tx\_Fridgemaster\_Domain\_Model\_Fridge) and we are still missing the object for Food. Let's activate this property (protected \$foods = array();) and create the Food class.

Food should have a property name and a property expiryDate. We start just with the property name.

---

**Example 4.18** Tx\_Fridgemaster\_Domain\_Model\_Food

---

```
/**
 * A Food item
 *
 * @version $Id:$
 * @copyright Copyright belongs to the respective authors
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License, version 2
 * @scope prototype
 * @entity
 */
class Tx_Fridgemaster_Domain_Model_Food extends Tx_Extbase_DomainObject_AbstractEntity {

    /**
     * The foods name
     *
     * @var string
     */
    protected $name = '';

    /**
     * Constructs this food item
     *
     * @return
     */
    public function __construct() {
    }

    /**
     * Sets this food's name
     *
     * @param string $name The food's name
     * @return void
     */
    public function setName($name) {
        $this->name = $name;
    }

    /**
     * Returns the food's name
     *
     * @return string The food's name
     */
    public function getName() {
        return $this->name;
    }
}
```

---

Now you should be able to see the food items you added in the backend.

Todo: screenshot from Backend?

We do not only want to add food from the backend but also from the frontend. We want to have this similar to the comments in `blog_example`. Let's implement that step by step:

modify our html template to have the necessary form:

---

**Example 4.19** show.html

```

<div class="csc-header csc-header-n1"><h1 class="csc-firstHeader">fridge: {fridge.location ←
  }</h1></div>
<ul>
  <f:for each="{fridge.foods}" as="food">
    <li>
      <p>
        {food.name}
      </p>
    </li>
  </f:for>
</ul>

<p>place more food into the fridge:</p>
<f:form method="post" controller="Food" action="add" name="newFood" object="{newFood}" ←
  arguments="{fridge : fridge}">
  <label for="name">Name of food</label>
  <f:form.textbox property="name" /><br /><br />
  <f:form.submit>Send</f:form.submit>
</f:form>

```

create the controller and action for handling the form data

**Example 4.20** addAction in Tx\_Fridgemaster\_Controller\_FoodController

we need to implement the addFood method in Tx\_Fridgemaster\_Domain\_Model\_Fridge

**Example 4.21** addFood

```

/**
 * Adds a food item to this fridge
 *
 * @param Tx_Fridgemaster_Domain_Model_Food $food
 * @return void
 */
public function addFood(Tx_Fridgemaster_Domain_Model_Food $food) {
    $this->foods[] = $food;
}

```

add the remove food link

**Example 4.22** removeFood

need foodRepository to autcreate food object, now dirty fridge object should be persisted automatically (currently not working with removed IRRE relation)

**4.5.5 add property expirydate to food**

fill field expirydate for at least one food in Backend

add property to Food (with getter and setter)

add property to template with templateHelper <f:format.date>

add form element to show.html

## 4.6

# **Part II**

# **In-Depth Manual**

---

## Chapter 5

# Introduction

This book is part of a bigger document about *a new way to write extensions*.

The whole document is structured into the following books:

- *Quickstart*: a hands-on example on how to write extensions with Extbase and Fluid
- *In-depth manual*: explaining all the details you need to know to use the system
- *Reference*: Containing all reference materials, conventions, ...
- *Developer reference*: Showing the framework internals for people wanting to get into framework programming

All books share the same introduction, so if this chapter looks familiar to you, just skip it!

### 5.1 A new way to write extensions

Extensions were introduced in TYPO3 3.5, and are one of the main reasons TYPO3 is so flexible, popular and widely used. However, the base class used for frontend extensions (called `tslib_pibase` - hence, we call old style extensions *pibase-style extensions*) has not changed a lot since its introduction a few years ago.

However, in the meantime many more advanced development concepts came into more widespread use in the PHP and TYPO3 community, especially the Model-View-Controller design pattern. We strongly feel the benefits of using such modern paradigms, as they provide more structure, guidance and help to the extension programmer and facilitate understanding foreign extensions.

All of the concepts introduced in this manual have been implemented by the TYPO3 v5 / FLOW3 team, and Extbase is a backport of the MVC and DDD functionality of FLOW3. We'd like to thank the many people involved in developing FLOW3 or helped with backporting the code to TYPO3 v4. Your input made this project a true community project.

### 5.2 Important terms

Here, we want to give some brief definitions of the most needed terms we'll use throughout the documents.

- **Model View Controller (MVC)**: A widely used design pattern which identifies three major concerns in every software project: A data model, a View to display the data to the user, and a Controller to handle user interaction and data flow.
  - **Domain Driven Design (DDD)**: A concept of structuring the Model. Its idea is to model the real world with objects, and implement both their behavior and data. It emphasizes the separation of the model and the repositories to access the model.
  - **FLOW3**: The next-generation enterprise PHP framework used as a basis to TYPO3 5.0
  - **Extbase**: A backport of the MVC and DDD functionality of FLOW3 to TYPO3 v4
  - **Fluid**: A templating engine designed for ease of use, flexibility and extensibility. It was specifically developed for FLOW3, and has been backported to TYPO3 v4 as well
-

## Chapter 6

# New extension structure

We decided to adjust the default directory structure of Extbase based extensions to resemble FLOW3 packages. Thus, the naming conventions shown here are the same as in FLOW3.

### 6.1 Directory structure

Every extension has the following subdirectories:

- *Classes/*  
All PHP class files
- *Resources/*
  - *Public/*  
Public resources, like images, css files
  - *Private/*
    - \* *Templates/*  
Fluid templates
- *Configuration/*  
All configuration which is not needed to be in the top level (tca.php, TypoScript, ...)
- *Documentation/*  
Manual (usually in DocBook format)
- *Tests/*  
Unit and integration tests

### 6.2 Class naming conventions

All classes must reside in the subdirectory `Classes/` in the extensions root. There is one class per file. We strongly encourage people to divide their tasks into many classes, thus you are advised to use subdirectories inside the `Classes/` subdirectory.

A class name consists of the following parts, separated by `_` (underscore):

- `Tx`: all TYPO3 classes following this naming convention start with `Tx`.
  - `ExtensionKey`: Upper-camel-cased extension key with underscores removed. Thus, the extension key `foo_bar` becomes `FooBar`.
-

- Path inside the classes directory: Everything after the extension key directly maps to the directory structure in the `Classes/` directory. If you replace every `_` by `/`, and append `.php`, you will have the file name for a given class.

A typical class name looks like `Tx_BlogExample_Controller_DefaultController`. Here, the extension key is `blog_example`, and you will find the PHP file where the class is defined in `blog_example/Classes/Controller/DefaultController.php`.

### 6.2.1 Interfaces

Interfaces end with `Interface` in their name and reside in the same directory structure as other classes. Example: `Tx_Extbase_MVC_Controller_ControllerInterface` resides in `extbase/Classes/MVC/Controller/ControllerInterface.php`

### 6.2.2 Abstract classes

Abstract classes start with `Abstract` in the last part of the class name. Example: `Tx_Extbase_MVC_Controller_AbstractController`, residing in `extbase/Classes/MVC/Controller/AbstractController.php`

---

## Chapter 7

# Extbase

### 7.1 Core concepts (Sebastian)

Here, we will explain the two main concepts of Extbase. The first one, Model View Controller, shortnamed MVC, is a design pattern which is used to structure applications on a high level. Its general idea is to split an application into three parts, namely Model, Controller and View.

The second concept called Domain Driven Design (DDD) deals with the Model and how to structure it.

#### 7.1.1 Model View Controller (MVC)

MVC is the core design pattern which splits an application into three parts: The *Model*, the *View*, and the *Controller*.

The *model* encapsulates all data used and manipulated by the application. The basic idea is to use *objects* as encapsulation for any data. See the next section about Domain Driven Design for more details on this.

The *view* renders data and deals with all the output logic. It gets some objects to render, and then the view knows *how* to render it.

The *controller* puts the the Model and View together: It reacts to user input, determines which data to load from the model and what view to use for rendering it. Each controller is subdivided into *actions*.

---

#### Example 7.1 MVC example

Let's take a car rental system as example: The model will consist of objects such as Bill, Car, Customer and SalesAgent which encapsulate the model's behavior and data. On the other hand, there might be a controller for dealing with Bills, with actions like *show* (list a single bill), *delete* (delete a bill), *edit* (edit a bill) or *new* (create a bill).

---

#### 7.1.2 Domain Driven Design (DDD)

Domain driven design gives some practices how you should design your application domain.

To explain this concept, we will use an example of an application where a car rental system should be designed.

The basic idea is to model every entity in the application domain (f.e. cars, bills, customers, sales agents, ...) as an object. We do not deal with databases or other persistence solutions directly, but we always work (and think) in objects. How these objects are persisted does not matter at all to us as an application developer - the framework handles this for us.

So, in the first step, it is a good idea to make a list of all objects in the application domain, which are usually the *nouns* in verbal descriptions. Then, collect *properties* of these objects - f.e. a car might have the following properties: *numberOfSeats*, *maxSpeed*, *color*, *licensePlate*. Now, think of *relations* between domain objects - here, it is really helpful to model your application domain as an UML class diagram.

---

Do not stop when you have collected the properties for your objects! Think of ways to *manipulate* objects in your application domain, and create methods on the corresponding objects for that. Example: A customer might need a method `rentCar(Car)`. However, make sure you will only put *domain-related methods* in there - methods like `show()`, `checkIfUserHasPermission()`, `export()`, ... are *not* belonging to your application domain because you cannot perform them in real life on the "real objects".

## 7.2 Storing data with domain models (Jochen)

## 7.3 Controller (Jochen / Sebastian)

A controller handles requests and generates a response to the user.

To generate a new controller, simply create a class which ends with `Controller` inside the `Classes/Controller` directory.

The controller is the main entry point for your application. A controller has to implement `Tx_Extbase_MVC_Controller_ControllerInterface`, but the most common controller is the `Tx_Extbase_MVC_Controller_ActionController`.

### 7.3.1 Actions

A controller has so-called *actions*, which are code-wise just public methods whose names end with `Action`. Think of actions as the ones doing the logic for fetching the data from the database (through repositories) and providing certain values for one specific request. If you want webpages in your app to list cars, view one single car, or edit a car, you'd have a "list" action, a "single" and an "edit" action.

---

#### Example 7.2 A very basic controller

---

```
class Tx_Blog_Controller_MyController {
    public function listAction() {...}
}
```

---

If no action is specified, the method named `indexAction` is called automatically, see `Tx_Extbase_MVC_Web_RequestBuilder::build()`.

### 7.3.2 Rendering a response

If an action returns something which is not `NULL`, then this will be returned to the user directly. If the controller does not return anything, then the `render()` method on the `View` is implicitly called.

---

#### Example 7.3 Rendering a response

---

```
class Tx_Blog_Controller_MyController {
    public function listAction() {
        return 'Hello world '; // Returns the string "Hello world " to the user
    }
    public function singleAction() {
        // no return -> the view is rendered by calling $this->view->render();
    }
}
```

---

Look into the section about the `View` below if you want to know how `Views` are resolved.

---

### 7.3.3 Arguments and validation

Actions would be pretty pointless if they could not react to user input. That's why they need *arguments* which can be set by the user.

There is one important rule different to most other frameworks: *All arguments must be registered!*<sup>1</sup> Thus, you need to specify explicitly which arguments you require and of what type they are.

Now - how can you register an argument? That is pretty simple: Just add them as method arguments to your action:

---

#### Example 7.4 Registering arguments

---

```
/**
 * @param $name string The name to greet❶
 */
public function greetAction($name) {❷
}

/**
 * @param $maxItems integer The maximum number of items to display
 */
public function listAction($maxItems = 20) {❸
}
```

Make sure to *always specify the PHPDoc* for the arguments. The data types are extracted from this definition and used to validate the input.

Arguments are just registered by specifying them as method parameters.

By default, arguments are mandatory. To specify *optional arguments*, just define a default parameter.

---

#### 7.3.3.1 Advanced validation

## 7.4 View (Sebastian)

A controller inheriting from `ActionController` (which most controllers do) uses the following strategy to resolve an appropriate View (for details see `Tx_Extbase_MVC_Controller_ActionController::resolveView()`):

- We start with the default view `Tx_Fluid_View_TemplateView` and check if there is a Fluid template available at `Resources/Private/Templates/ControllerName/actionname.html`. If yes, we use it.
- We use `$this->viewObjectNamePattern` (which is set to `Tx_@extension_View_@controller_@action` by default) to resolve an individual view. If there is a class with the specified name, we use it.
- Else, we use `Tx_Extbase_MVC_View_EmptyView`.

All PHP based views have to implement `Tx_Extbase_MVC_View_ViewInterface`.

For more information about Fluid templates, have a look at the Fluid chapter below.

## 7.5 FAQ / Best practices

---

<sup>1</sup>Never call `$_GET` or `$_POST` directly! This is strictly forbidden!

---

## Chapter 8

# Fluid (Sebastian)

This chapter describes all things that a developer that is writing code in this 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, and finishes with how to write your own view helpers.

### 8.1 Basic concepts

This section describes all basic concepts available.

This includes:

- Variables / Object Accessors
- View Helpers
- Arrays

#### 8.1.1 Variables and Object Accessors

A templating system would be quite pointless if it were 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 just write the following snippet in your controller:

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

Then, you simply output the blog title in your template with the following line:

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

Now, if you want to extend the output by the blog author as well, just repeat the steps above, but imagine having several variables of a blog post that you want to render, then it would be quite inconvenient and hard to read.<sup>1</sup>

That's why the template language has a special syntax for object access, as 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>
```

---

<sup>1</sup>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.

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()`.

---

## 8.1.2 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:

---

### Example 8.1 Tags and Namespace declarations

---

```
{namespace f=Tx_Fluid_ViewHelpers}&#x2044;❶
<f:link.action controller="Administration">Administration</f:link.action>❷
```

*Namespace Declaration:* You import the PHP Namespace `Tx_Fluid_ViewHelpers` under the prefix `f`. Hint: you can leave out this namespace import because it is imported by default.<sup>2</sup>

*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 `<f:link.action />` tag is implemented in the class `Tx_Fluid_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 `Tx_Fluid_ViewHelpers`)
- The unqualified name of the tag, without the prefix, is capitalized (`Link`), and the postfix `ViewHelper` is appended. As you see in our example you can group view helpers (put them together into specific directories) using the same naming schema as classes in Extbase and calling them in your template as `[groupname].[view helper name]`.

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 will include Eclipse autocompletion<sup>3</sup>

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.

---

<sup>3</sup>This is done through XML Schema Definition files which are generated from the view helper's PHPdoc comments.

---

This is often used when adding arguments to links:

---

**Example 8.2** Creating a link with arguments
 

---

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

---

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.

---

### 8.1.2.1 Boolean expressions

Often, you need some kind of conditions inside your template. For them, you will usually use the `<f:if>` ViewHelper. This ViewHelper has an argument `condition` of type boolean. The following two syntaxes are available for all arguments of type boolean:

#### 8.1.2.1.1 Simple boolean evaluations

You can just pass any value into a boolean argument, which will then be evaluated according to the following conditions:

- If it is *boolean*, the boolean value is just taken.
- If it is *string* and the string is "false" or empty, FALSE is returned. If the string is not empty and not "false", TRUE is returned.
- If it is *numeric*, then it is TRUE if it is greater than 0
- If it is *array* or a *Countable* object, then it is TRUE if `count($object) > 0`.
- If it is *object*, then TRUE is returned.
- Else, FALSE is returned.

---

**Example 8.3** Using boolean evaluations
 

---

```
<f:if condition="{post.numberOfComments}">Only shows if there is at least one comment</f:if ←
>
```

---

#### 8.1.2.1.2 Complex boolean expressions

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 at how this works:

**Example 8.4** Using boolean expressions

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

```
<f:if condition="{myNumber} == 0">...</f:if>
```
  - A JSON Array
  - A `ViewHelper`

```
<f:if condition="{post} == {blog:mySpecialViewHelper()}">...</f:if>
```
  - An Object Accessor (this is probably the most used example)
 

```
&#x2044;<f:if condition="{post} == {currentBlogPosting}">...</f:if>
```

**8.1.3 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<sup>4</sup>. 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 quotes)"
}
```

- a nested array

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

<sup>4</sup>Actually, it should be the same. If not, please tell us!

- a variable reference (=an object accessor)

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

## 8.2 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.

## 8.3 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 `Tx_Fluid_Core_AbstractViewHelper` (or its subclasses). You need to implement only one method to write a view helper:

```
public function render()
```

### 8.3.1 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 `Tx_Blog_ViewHelpers`), Fluid will automatically instantiate the class `Tx_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:

- `$this->renderChildren()` renders everything between the opening and closing tag of the view helper and returns the rendered result (as string).
- `$this->templateVariableContainer` is an instance of `Tx_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->templateVariableContainer->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 `TemplateVariableContainer` if a variable of the same name already exists - again to prevent side effects and scope problems.

- `$this->arguments` is a read-only associative array where you will find the values for all arguments you registered previously. Normally you do not need it, please have a look at how argument registration works below.

Now, we will look at an example: How to write a view helper giving us the `foreach` functionality of PHP.<sup>5</sup>

---

<sup>5</sup>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.

**Example 8.5** Implementing a loop

A loop should 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*<sup>6</sup> 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 Tx_Fluid_ViewHelpers_ForViewHelper {
  /**
   * Renders a loop
   *
   * @param array $each Array to iterate over1
   * @param string $as Iteration variable
   */
  public function render(array $each, $as) {2
    $out = '';
    foreach ($each as $singleElement) {
      $this->templateVariableContainer->add($as, $singleElement);
      $out .= $this->renderChildren();3
      $this->templateVariableContainer->remove($as);4
    }
    return $out;
  }
}
```

The PHPDoc *is part of the code!* Fluid extracts the argument datatypes from the PHPDoc.

You can simply register arguments to the view helper by adding them as method arguments of the `render()` method and adding PHPDoc for these arguments.

Here, everything between the opening and closing tag of the view helper is rendered and returned as string.

Make sure to remove the variable again to prevent side-effects!

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.

### 8.3.2 Declaring arguments

There are two ways to declare and access arguments:

- Method arguments are ViewHelper arguments
- `initializeArguments()`

### 8.3.2.1 render() method parameters are ViewHelper arguments

If the render() method has some parameters, they will be automatically registered as ViewHelper arguments. However, *there must be PHPDoc for these parameters!* We need the type information from them. Let's have another look at some examples:

#### Example 8.6 render() method parameters

```
/**&#x2028;
 * @param string $email The email to process&#x2028;
 */
&#x2028;public function render($email) {&#x2028;
    // do something with the email
&#x2028;}
```

The PHPDoc *is part of the code!* Fluid extracts the argument datatypes from the PHPDoc.

You can simply register arguments to the view helper by adding them as method arguments of the render() method and adding PHPDoc for these arguments.

### 8.3.2.2 initializeArguments() for argument initialization

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().

To access arguments you registered in initializeArguments(), use the read-only array \$this->arguments[...].

### 8.3.3 TagBasedViewHelper

Many view helpers output an HTML tag - for example <f: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:

```
<?php
class Tx_Fluid_ViewHelpers_Link_ActionViewHelper extends Tx_Fluid_Core_ViewHelper_TagBasedViewHelper {

    protected $tagName = 'a';

    /**
     * Arguments initialization
     */
    public function initializeArguments() {
        $this->registerUniversalTagAttributes();
        $this->registerTagAttribute('target', 'string', 'Target of link', FALSE);
        $this->registerTagAttribute('rel', 'string', 'Specifies the relationship between the
            current document and the linked document', FALSE);
    }

    /**
     * @param string $action Target action
     * @param array $arguments Arguments
     * @param string $controller Target controller. If NULL current controllerName is used
     * @param string $extensionName Target Extension Name (without "tx_" prefix and no
        underscores). If NULL the current extension name is used
     * @param string $pluginName Target plugin. If empty, the current plugin name is used
     * @param integer $page target page. See TypoLink destination
     * @param integer $pageType type of the target page. See typolink.parameter
     * @param boolean $noCache set this to disable caching for the target page. You should
        not need this.
     * @param boolean $noCacheHash set this to suppress the cHash query parameter created by
        TypoLink. You should not need this.
     * @param string $section the anchor to be added to the URI
     * @param boolean $linkAccessRestrictedPages If set, links pointing to access restricted
        pages will still link to the page even though the page cannot be accessed.
     * @param array $additionalParams additional query parameters that won't be prefixed like
        $arguments (override $arguments)
     * @return string Rendered link
     */
    public function render($action, array $arguments = array(), $controller = NULL,
        $extensionName = NULL, $pluginName = NULL, $pageUid = NULL, $pageType = 0, $noCache =
        FALSE, $noCacheHash = FALSE, $section = '', $linkAccessRestrictedPages = FALSE, array
        $additionalParams = array()) {
        $URIBuilder = $this->controllerContext->getURIBuilder();
        $uri = $URIBuilder->URIFor($pageUid, $action, $arguments, $controller, $extensionName,
            $pluginName, $pageType, $noCache, !$noCacheHash, $section,
            $linkAccessRestrictedPages, $additionalParams);

        $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.

### 8.3.3.1 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.<sup>7</sup> 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 set</ ←
  f:link.action>

```

This attribute is available in all tags that inherit from `Tx_Fluid_Core_ViewHelper_TagBasedViewHelper`.

## 8.3.4 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! They will probably still change!!*

### 8.3.4.1 SubNodeAccess Facet

*Currently, all facets are NOT YET PUBLIC API! Use them at your own risk! They will probably still change!!*

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:

<sup>7</sup>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)

```
<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.

### 8.3.4.2 PostParse Facet

*Currently, all facets are NOT YET PUBLIC API! Use them at your own risk! They will probably still change!!*

---

#### 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 $node,
  $syntaxTreeNode, $viewHelperArguments, \F3\Fluid\Core\ViewHelper\
  TemplateVariableContainer $variableContainer);
```

Note this method is *static*<sup>8</sup>. 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`.

---

<sup>8</sup>It is static because the `ViewHelper` has not been instantiated at that point in time, thus it is forbidden to set any instance variables.

---

## **Part III**

# **Reference**

---

## Chapter 9

# Introduction

This book is part of a bigger document about *a new way to write extensions*.

The whole document is structured into the following books:

- *Quickstart*: a hands-on example on how to write extensions with Extbase and Fluid
- *In-depth manual*: explaining all the details you need to know to use the system
- *Reference*: Containing all reference materials, conventions, ...
- *Developer reference*: Showing the framework internals for people wanting to get into framework programming

All books share the same introduction, so if this chapter looks familiar to you, just skip it!

### 9.1 A new way to write extensions

Extensions were introduced in TYPO3 3.5, and are one of the main reasons TYPO3 is so flexible, popular and widely used. However, the base class used for frontend extensions (called `tslib_pibase` - hence, we call old style extensions *pibase-style extensions*) has not changed a lot since its introduction a few years ago.

However, in the meantime many more advanced development concepts came into more widespread use in the PHP and TYPO3 community, especially the Model-View-Controller design pattern. We strongly feel the benefits of using such modern paradigms, as they provide more structure, guidance and help to the extension programmer and facilitate understanding foreign extensions.

All of the concepts introduced in this manual have been implemented by the TYPO3 v5 / FLOW3 team, and Extbase is a backport of the MVC and DDD functionality of FLOW3. We'd like to thank the many people involved in developing FLOW3 or helped with backporting the code to TYPO3 v4. Your input made this project a true community project.

### 9.2 Important terms

Here, we want to give some brief definitions of the most needed terms we'll use throughout the documents.

- **Model View Controller (MVC)**: A widely used design pattern which identifies three major concerns in every software project: A data model, a View to display the data to the user, and a Controller to handle user interaction and data flow.
  - **Domain Driven Design (DDD)**: A concept of structuring the Model. Its idea is to model the real world with objects, and implement both their behavior and data. It emphasizes the separation of the model and the repositories to access the model.
  - **FLOW3**: The next-generation enterprise PHP framework used as a basis to TYPO3 5.0
  - **Extbase**: A backport of the MVC and DDD functionality of FLOW3 to TYPO3 v4
  - **Fluid**: A templating engine designed for ease of use, flexibility and extensibility. It was specifically developed for FLOW3, and has been backported to TYPO3 v4 as well
-

## **Chapter 10**

# **Extbase (Jochen)**

### **10.1 Caching**

## Chapter 11

# Fluid (Sebastian)

### 11.1 Standard View Helper Library

Should be autogenerated from the tags.

#### 11.1.1 alias

Alias view helper

##### 11.1.1.1 Examples

---

**Example 11.1** Single alias

---

```
<f:alias map="{x: 'foo'}">{x}</f:alias>
```

---

Output:

foo

---

**Example 11.2** Multiple mappings

---

```
<f:alias map="{x: foo.bar.baz, y: foo.bar.baz.name}">
  {x.name} or {y}
</f:alias>
```

---

Output:

[name] or [name]

depending on {foo.bar.baz}

##### 11.1.1.2 Arguments

#### 11.1.2 base

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

---

Name	Type	Required	Description
map	array	yes	

Table 11.1: Arguments

### 11.1.2.1 Examples

#### Example 11.3 Example

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

Output:

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

(depending on your domain)

### 11.1.2.2 Arguments

No arguments defined.

## 11.1.3 cObject

This class is a TypoScript view helper for the Fluid templating engine.

### 11.1.3.1 Arguments

Name	Type	Required	Description
tyoscriptObjectPath	string	yes	the TypoScript setup path of the TypoScript object to render
currentValueKey	string	no	

Table 11.2: Arguments

## 11.1.4 debug

### 11.1.4.1 Arguments

Name	Type	Required	Description
title	string	no	

Table 11.3: Arguments

## 11.1.5 else

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

### 11.1.5.1 Arguments

No arguments defined.

## 11.1.6 for

Loop view helper

### 11.1.6.1 Examples

---

#### Example 11.4 Simple

---

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

---

Output:

1234

---

#### Example 11.5 Output array key

---

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

### 11.1.6.2 Arguments

Name	Type	Required	Description
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

Table 11.4: Arguments

## 11.1.7 form

Form view helper. Generates a <form> Tag.

---

---

### 11.1.7.1 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 11.6 Example

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

---

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

---

#### Example 11.7 Form with enctype set

```
<f:form action=".." controller="..." package="..." enctype="multipart/form-data">...</f: ↵  
form>
```

---

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

---

#### Example 11.8 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.

### 11.1.7.4 Arguments

## 11.1.8 form.hidden

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

### 11.1.8.1 Examples

---

#### Example 11.9 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.

### 11.1.8.2 Arguments

## 11.1.9 form.select

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

---

Name	Type	Required	Description
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
controller	string	no	Target controller
extensionName	string	no	Target Extension Name (without "tx_" prefix and no underscores). If NULL the current extension name is used
pluginName	string	no	Target plugin. If empty, the current plugin name is used
pageUid	integer	no	Target page uid
options	array	no	typolink options
object	mixed	no	Object to use for the form. Use in conjunction with the "property" attribute on the sub tags
pageType	integer	no	Target page type
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. 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

Table 11.5: Arguments

Name	Type	Required	Description
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. 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

Table 11.6: Arguments

### 11.1.9.1 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.

---

**Example 11.10** Basic usage

---

```
<f3:form.select name="paymentOptions" options="{paypal: 'PayPal International Services',  
  visa: 'VISA Card'}" />
```

---

### 11.1.9.2 Pre-select a value

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

---

**Example 11.11** Default value

---

```
<f3:form.select name="paymentOptions" options="{paypal: 'PayPal International Services',  
  visa: 'VISA Card'}" selectedValue="visa" />
```

---

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.

### 11.1.9.3 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.

---

**Example 11.12** 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.

### 11.1.9.4 Arguments

### 11.1.10 form.submit

Creates a submit button.

---

<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Description</b>
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. 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
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.

Table 11.7: Arguments

### 11.1.10.1 Examples

---

#### Example 11.13 Defaults

---

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

---

Output:

```
<input type="submit" />
```

---

#### Example 11.14 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" />
```

### 11.1.10.2 Arguments

Name	Type	Required	Description
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. 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

Table 11.8: Arguments

### 11.1.11 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.

---

### 11.1.11.1 Examples

#### Example 11.15 Example

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

Output:

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

### 11.1.11.2 Arguments

Name	Type	Required	Description
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. 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

Table 11.9: Arguments

### 11.1.12 form.textbox

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

### 11.1.12.1 Examples

#### Example 11.16 Example

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

Output:

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

### 11.1.12.2 Arguments

Name	Type	Required	Description
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. 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

Table 11.10: Arguments

### 11.1.13 form.upload

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

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

### 11.1.13.1 Examples

#### Example 11.17 Example

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

Output:

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

### 11.1.13.2 Arguments

Name	Type	Required	Description
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. 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

Table 11.11: Arguments

### 11.1.14 format.crop

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

### 11.1.14.1 Examples

---

#### Example 11.18 Defaults

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

---

Output:

This is...

---

#### Example 11.19 Custom suffix

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

---

Output:

This is some&nbsp;[more]

---

#### Example 11.20 Don't respect word boundaries

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

---

Output:

This is so...

WARNING: This tag does NOT handle tags currently.

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

### 11.1.14.2 Arguments

Name	Type	Required	Description
maxCharacters	integer	yes	Place where to truncate the string
append	string	no	What to append, if truncation happened
respectWordBoundaries	boolean	no	If TRUE and division is in the middle of a word, the remains of that word is removed

Table 11.12: Arguments

### 11.1.15 format.currency

Formats a given float to a currency representation.

---

### 11.1.15.1 Examples

---

#### Example 11.21 Defaults

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

---

Output:

123,46

---

#### Example 11.22 All parameters

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

---

Output:

54,321.00 \$

### 11.1.15.2 Arguments

Name	Type	Required	Description
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.

Table 11.13: Arguments

### 11.1.16 format.date

Formats a DateTime object.

#### 11.1.16.1 Examples

---

#### Example 11.23 Defaults

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

---

Output:

1980-12-13

(depending on the current date)

---

#### Example 11.24 Custom date format

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

---

Output:

01:23

(depending on the current time)

---

#### Example 11.25 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 strtotime.php>)

#### 11.1.16.2 Arguments

Name	Type	Required	Description
format	string	no	Format String which is taken to format the Date/Time

Table 11.14: Arguments

#### 11.1.17 format.html

Renders a string by passing it to a TYPO3 parseFunc.

You can either specify a path to the TypoScript setting or set the parseFunc options directly.

By default lib.parseFunc\_RTE is used to parse the string.

Example:

(1) default parameters:

```
<f:format.html>foo <b>bar</b>. Some <LINK 1>link</LINK>.</f:format.html>
```

Result:

```
<p class="bodytext">foo <b>bar</b>. Some <a href="index.php?id=1" >link</a>.</p>
```

(depending on your TYPO3 setup)

(2) custom parseFunc

```
<f:format.html parseFuncTSPath="lib.parseFunc">foo <b>bar</b>. Some <LINK 1>link</LINK>.</f:format.html>
```

Output:

```
foo <b>bar</b>. Some <a href="index.php?id=1" >link</a>.
```

#### 11.1.17.1 Arguments

#### 11.1.18 format.nl2br

Wrapper for PHP's nl2br function.

---

Name	Type	Required	Description
parseFuncTSPath	string	no	path to TypoScript parseFunc setup.

Table 11.15: Arguments

### 11.1.18.1 Arguments

No arguments defined.

## 11.1.19 format.number

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

### 11.1.19.1 Arguments

Name	Type	Required	Description
decimals	int	no	The number of digits after the decimal point
decimalSeparator	string	no	The decimal point character
thousandsSeparator	string	no	The character for grouping the thousand digits

Table 11.16: Arguments

## 11.1.20 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>

### 11.1.20.1 Examples

---

#### Example 11.26 Scientific notation

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

Output:

3.625e+8

---

#### Example 11.27 Argument swapping

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

Output:

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

---

**Example 11.28** Single argument

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

Output:

We love TYPO3

**11.1.20.2 Arguments**

Name	Type	Required	Description
arguments	array	yes	The arguments for vsprintf

Table 11.17: Arguments

**11.1.21 if**

This view helper implements an if/else condition.

**11.1.21.1 Arguments**

Name	Type	Required	Description
condition	boolean	yes	View helper condition

Table 11.18: Arguments

**11.1.22 image****11.1.22.1 Arguments****11.1.23 link.action**

A view helper for creating links to extbase actions.

**11.1.23.1 Examples****Example 11.29** link to the show-action of the current controller

```
<f:link.action action="show">action link</f:link.action>
```

Output:

```
<a href="index.php?id=123&tx_myextension_plugin[action]=show&tx_myextension_plugin[controller]=Standard&cHash=xyz">action link</f:link.action>
```

(depending on the current page and your TS configuration)

Name	Type	Required	Description
src	string	yes	
width	string	no	width of the image. This can be a numeric value representing the fixed width of the image in pixels. But you can also perform simple calculations by adding "m" or "c" to the value. See <code>imgResource.width</code> for possible options.
height	string	no	height of the image. This can be a numeric value representing the fixed height of the image in pixels. But you can also perform simple calculations by adding "m" or "c" to the value. See <code>imgResource.width</code> for possible options.
minWidth	integer	no	minimum width of the image
minHeight	integer	no	minimum height of the image
maxWidth	integer	no	maximum width of the image
maxHeight	integer	no	maximum height of the image
class	string	no	CSS class(es) for this element
dir	string	no	Text direction for this HTML element. 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
alt	string	yes	Specifies an alternate text for an image
ismap	string	no	Specifies an image as a server-side image-map. Rarely used. Look at <code>usemap</code> instead
longdesc	string	no	Specifies the URL to a document that contains a long description of an image
usemap	string	no	Specifies an image as a client-side image-map

Table 11.19: Arguments

Name	Type	Required	Description
additionalAttributes	array	no	Additional tag attributes. They will be added directly to the resulting HTML tag.
action	string	yes	Target action
arguments	array	no	Arguments
controller	string	no	Target controller. If NULL current controllerName is used
extensionName	string	no	Target Extension Name (without "tx_" prefix and no underscores). If NULL the current extension name is used
pluginName	string	no	Target plugin. If empty, the current plugin name is used
pageUid	integer	no	target page. See TypoLink destination
pageType	integer	no	type of the target page. See typolink.parameter
noCache	boolean	no	set this to disable caching for the target page. You should not need this.
noCacheHash	boolean	no	set this to suppress the cHash query parameter created by TypoLink. You should not need this.
section	string	no	the anchor to be added to the URI
linkAccessRestrictedPages	boolean	no	If set, links pointing to access restricted pages will still link to the page even though the page cannot be accessed.
additionalParams	array	no	additional query parameters that won't be prefixed like \$arguments (overrule \$arguments)
class	string	no	CSS class(es) for this element
dir	string	no	Text direction for this HTML element. 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

### 11.1.23.2 Arguments

### 11.1.24 link.email

Email link view helper.

Generates an email link incorporating TYPO3s spamProtectEmailAddresses-settings.

= Examples

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

#### 11.1.24.1 Arguments

Name	Type	Required	Description
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.

Table 11.21: Arguments

### 11.1.25 link.external

A view helper for creating links to external targets.

#### 11.1.25.1 Examples

---

**Example 11.30** 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>
```

#### 11.1.25.2 Arguments

### 11.1.26 link.page

A view helper for creating links to TYPO3 pages.

#### 11.1.26.1 Examples

---

**Example 11.31** link to the current page

---

```
<f:link.page>page link</f:link.page>
```

---

Output:

---

<b>Name</b>	<b>Type</b>	<b>Required</b>	<b>Description</b>
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. 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

Table 11.22: Arguments

```
<a href="index.php?id=123">page link</f:link.action>
```

(depending on the current page and your TS configuration)

---

**Example 11.32** query parameters

---

```
<f:link.page pageUId="1" additionalParams="{foo: 'bar'}">page link</f:link.page>
```

---

Output:

```
<a href="index.php?id=1&foo=bar">page link</f:link.action>
```

(depending on your TS configuration)

### 11.1.26.2 Arguments

### 11.1.27 then

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

### 11.1.27.1 Arguments

No arguments defined.

### 11.1.28 translate

Translate a key from locallang. The files are loaded from the folder "Resources/Private/Language/".

### 11.1.28.1 Arguments

### 11.1.29 uri.action

A view helper for creating URIs to extbase actions.

### 11.1.29.1 Examples

---

**Example 11.33** URI to the show-action of the current controller

---

```
<f:uri.action action="show" />
```

---

Output:

```
index.php?id=123&tx_myextension_plugin[action]=show&tx_myextension_plugin[controller]=Standard&cHash=xyz
```

(depending on the current page and your TS configuration)

---

Name	Type	Required	Description
additionalAttributes	array	no	Additional tag attributes. They will be added directly to the resulting HTML tag.
pageUid	integer	no	target page. See TypoLink destination
additionalParams	array	no	query parameters to be attached to the resulting URI
pageType	integer	no	type of the target page. See <code>typolink.parameter</code>
noCache	boolean	no	set this to disable caching for the target page. You should not need this.
noCacheHash	boolean	no	set this to suppress the <code>cHash</code> query parameter created by TypoLink. You should not need this.
section	string	no	the anchor to be added to the URI
linkAccessRestrictedPages	boolean	no	If set, links pointing to access restricted pages will still link to the page even though the page cannot be accessed.
class	string	no	CSS class(es) for this element
dir	string	no	Text direction for this HTML element. 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

Table 11.23: Arguments

Name	Type	Required	Description
key	string	yes	The locallang key
htmlEscape	boolean	no	TRUE if the result should be htmlentities

Table 11.24: Arguments

Name	Type	Required	Description
action	string	yes	Target action
arguments	array	no	Arguments
controller	string	no	Target controller. If NULL current controllerName is used
extensionName	string	no	Target Extension Name (without "tx_" prefix and no underscores). If NULL the current extension name is used
pluginName	string	no	Target plugin. If empty, the current plugin name is used
pageUid	integer	no	target page. See TypoLink destination
pageType	integer	no	type of the target page. See typolink.parameter
noCache	boolean	no	set this to disable caching for the target page. You should not need this.
noCacheHash	boolean	no	set this to suppress the cHash query parameter created by TypoLink. You should not need this.
section	string	no	the anchor to be added to the URI
linkAccessRestrictedPages	boolean	no	If set, links pointing to access restricted pages will still link to the page even though the page cannot be accessed.
additionalParams	array	no	additional query parameters that won't be prefixed like \$arguments (override \$arguments)

Table 11.25: Arguments

### 11.1.29.2 Arguments

### 11.1.30 uri.email

Email URI view helper.

Generates an email URI incorporating TYPO3s spamProtectEmailAddresses-settings.

= Examples

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

#### 11.1.30.1 Arguments

Name	Type	Required	Description
email	string	yes	The email address to be turned into a URI

Table 11.26: Arguments

### 11.1.31 uri.external

A view helper for creating URIs to external targets.

Currently the specified URI is simply passed through.

#### 11.1.31.1 Examples

---

#### Example 11.34 Example

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

---

Output:

http://www.typo3.org

#### 11.1.31.2 Arguments

Name	Type	Required	Description
uri	string	yes	the target URI

Table 11.27: Arguments

### 11.1.32 uri.page

A view helper for creating URIs to TYPO3 pages.

---

### 11.1.32.1 Examples

---

#### Example 11.35 URI to the current page

---

```
<f:uri.page>page link</f:uri.page>
```

---

Output:

index.php?id=123

(depending on the current page and your TS configuration)

---

#### Example 11.36 query parameters

---

```
<f:uri.page pageUid="1" additionalParams="{foo: 'bar'}" />
```

---

Output:

index.php?id=1&foo=bar

(depending on your TS configuration)

### 11.1.32.2 Arguments

Name	Type	Required	Description
pageUid	integer	no	target PID
additionalParams	array	no	query parameters to be attached to the resulting URI
pageType	integer	no	type of the target page. See <code>typolink.parameter</code>
noCache	boolean	no	set this to disable caching for the target page. You should not need this.
noCacheHash	boolean	no	set this to suppress the <code>cHash</code> query parameter created by <code>TypoLink</code> . You should not need this.
section	string	no	the anchor to be added to the URI
linkAccessRestrictedPages	boolean	no	If set, links pointing to access restricted pages will still link to the page even though the page cannot be accessed.

Table 11.28: Arguments

---

## **Part IV**

# **Framework reference**

---

## Chapter 12

# Introduction

This book is part of a bigger document about *a new way to write extensions*.

The whole document is structured into the following books:

- *Quickstart*: a hands-on example on how to write extensions with Extbase and Fluid
- *In-depth manual*: explaining all the details you need to know to use the system
- *Reference*: Containing all reference materials, conventions, ...
- *Developer reference*: Showing the framework internals for people wanting to get into framework programming

All books share the same introduction, so if this chapter looks familiar to you, just skip it!

### 12.1 A new way to write extensions

Extensions were introduced in TYPO3 3.5, and are one of the main reasons TYPO3 is so flexible, popular and widely used. However, the base class used for frontend extensions (called `tslib_piBase` - hence, we call old style extensions *piBase-style extensions*) has not changed a lot since its introduction a few years ago.

However, in the meantime many more advanced development concepts came into more widespread use in the PHP and TYPO3 community, especially the Model-View-Controller design pattern. We strongly feel the benefits of using such modern paradigms, as they provide more structure, guidance and help to the extension programmer and facilitate understanding foreign extensions.

All of the concepts introduced in this manual have been implemented by the TYPO3 v5 / FLOW3 team, and Extbase is a backport of the MVC and DDD functionality of FLOW3. We'd like to thank the many people involved in developing FLOW3 or helped with backporting the code to TYPO3 v4. Your input made this project a true community project.

### 12.2 Important terms

Here, we want to give some brief definitions of the most needed terms we'll use throughout the documents.

- **Model View Controller (MVC)**: A widely used design pattern which identifies three major concerns in every software project: A data model, a View to display the data to the user, and a Controller to handle user interaction and data flow.
- **Domain Driven Design (DDD)**: A concept of structuring the Model. Its idea is to model the real world with objects, and implement both their behavior and data. It emphasizes the separation of the model and the repositories to access the model.
- **FLOW3**: The next-generation enterprise PHP framework used as a basis to TYPO3 5.0
- **Extbase**: A backport of the MVC and DDD functionality of FLOW3 to TYPO3 v4
- **Fluid**: A templating engine designed for ease of use, flexibility and extensibility. It was specifically developed for FLOW3, and has been backported to TYPO3 v4 as well