open source PRESS

TYPO3 Extbase

Modern Extension Development for TYPO3 CMS with Extbase & Fluid

professional reference

Patrick Lobacher · Michael Schams

Table of Contents

14. Multi-Language

Patrick Lobacher

Michael Schams

Modern Extension Development for TYPO3 CMS with Extbase & Fluid

Open Source Press

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.dnb.de.

Copyright © 2015 Open Source Press GmbH
Editors: Gemma Creegan, Andrew Walters
Typesetting: textovia web application (textovia.com)
Graphic Designer: Lena Levitina
ISBN: 9783955391546 (E-Book Mobi) http://www.opensourcepress.de

Preface

Do feel free to be excited because first and foremost, Extbase and Fluid are one thing: fun!

If you are experienced with software development, you will soon realise that a lot of features in Extbase and Fluid are simply intuitive or can be learnt quickly. After working through this book, you will be able to develop large projects with Extbase and Fluid and in fact, it will be a piece of cake.

Before we start, it is a good idea to bookmark the offical Internet address of this book: http://www.extbase-book.org.

You can not only access some of the files listed in this book, but we try to keep this site up-to-date when a breaking change happens in TYPO3, which affects the workflow described and was unpredictable at the publishing date.

1. Patrick’s Background

It was partly my fault, when TYPO3 version 4.3 was released in 2009 and featured Extbase and Fluid back then. Oliver Hader – Release Manager of this TYPO3 version – approached us during the T3CON09 in Frankfurt/Germany and asked about our opinion, if the next version of TYPO3 CMS should include either Extbase and Fluid or another feature (which I do not remember any more). My answer was immediate: the TYPO3 community needs Extbase and Fluid – as soon as possible!

Back then, we experimented with the pre-alpha-version of Extbase and Fluid in our agency and were totally convinced of this powerful technology. No doubt, this version still included numerous bugs and shortcomings but it became obvious that this could be the way forward for the TYPO3 project.

As a matter of fact, TYPO3 version 4.3 was released in December 2009 and was shipped with Extbase and Fluid.

As early as June 2009, I made the decision to write about this topic and devoted the third part of my book “TYPO3 Extensions – Professional Frontend- and Backend Development” (published by Hanser Verlag) Extbase and Fluid without further ado. I gathered all relevant information that were important at the time. Therefore the book, published in May 2010, contributed to the growing popularity of the technology.

A second book, written by the “fathers” of Extbase and Fluid – Jochen Rau and Sebastian Kurfürst – was published in July 2010 and I am still proud of the fact, that they allowed me to contribute the OOP (Object Orientated Programming) introduction chapter for this.

Since then, no further books have been published although the demand for new and detailed information about Extbase and Fluid is still high. Especially after the release of TYPO3 CMS version 6.2 because Extbase and Fluid will be used and maintained for at least another three years: version 6.2 is a “long-term-support” release (LTS).

With the release of the LTS version of TYPO3 CMS 7 (scheduled for late 2015), this time period will even be extended.

Therefore I made the decision to write another book about this topic.

My former agency – the typovision GmbH in Munich – has used Extbase and Fluid since 2009 for all its projects since then. There was not a single time when we regretted this decision although we Extbase and Fluid do have their issues at times. As everyone knows, no technology is flawless and we are glad that we parted with the classical method of extension development long ago.

Over the last three years, I conducted about 100 seminars about this topic and published approximately 10 articles and two books (including this one). In addition, I tried to inspire everyone who was interested in Extbase and Fluid at numerous events. As a consequence, I was nicknamed “Extbase Evangelist” for quite a long time and I still show my enthusiasm for this technology today. My single purpose is to spread the Extbase virus because this is definitely the future of extension development in TYPO3 and therefore the success of this extraordinary enterprise content management system which is rightly one of the market leaders of its kind today.

2. Michael’s Background

I have been working with TYPO3 CMS since 2004. My first encounter with this CMS was version 3.6 but while I was curious about using it, I did not like it very much to be honest. Using TYPO3 then does not compare with what you get today.

TYPO3 has became much easier to handle, to configure and to understand over time. Everyone active in the TYPO3 community (editors, integrators, administrators as well as developers) helped transform a stubborn beast into a real enterprise content management system, without loosing all its powerful features. In addition, TYPO3’s flexibility is awesome: compared with other PHP-based content management systems, extension developers can do whatever their heart’s content – in a professional way and without the need to “hack” the core of the system.

However there are two major drawbacks from my perspective. The first is that the TYPO3 community is mainly located in Europe, especially Germany – which is a distance of approximately 16,000 km from where I currently live: Melbourne (Australia). The problem is not only to build a community in Australia but also the fact that a lot of resources are only in German and this does not only affect Australia but every non-German-speaking TYPO3 enthusiast worldwide (including Europe).

The second “issue” with TYPO3 is – and this is not only TYPO3-specific (in fact, we see this in many open source content management systems) – the code base of extensions often lacks quality, security and consistency.

The clarity of Extbase, the modern standards and programming principles, the way developers must base their code on Extbase standards and the fact, that developers should think about and plan their approach before one line of code is even written. All these factors directly address the issues we have seen over the last few years but the best is that writing TYPO3 extensions in Extbase also saves a lot of time.

Mainly these two points encouraged me to work on the English edition of the Extbase book when Patrick asked me if I would be interested. Hopefully our work will support the non-German TYPO3 community to develop interesting TYPO3 extensions and to publish them proudly in the official TYPO3 Extension Repository (TER).

3. Acknowledgement

We would like to thank all readers: it is our pleasure to transfer our knowledge and experience in a book because we sourced most of our expertise from high quality books as well.

Many, many thanks to: Anja Leichsenring, Nicole Cordes, Matthias Schröder, Stefan Frömken and Stefan Völker.

We would also like to thank our two English language reviewers, who did a fantastic job. Without their help, this book would not be possible: Gemma Creegan and Andrew Walters.

But now, let’s focus on the book and start learning Extbase and Fluid!

Patrick Lobacher and Michael Schams

April 2015

Chapter 1. The Modern Way of Extension Development

In 2006, during the first *TYPO3 Developer Days* (T3DD06), a decision was made to re-develop the content management framework TYPO3 from scratch.[1] This was based on the consensus that the existing system would not be able to meet future requirements.

1.1. A New TYPO3 Arises

As version 4.0 was already released, a version number for the new CMS was determined quickly: TYPO3 5.0 code name “Phoenix” – in the style of the mythical bird, which obtains new life by rising from the ashes of its predecessor. A *TYPO3 5.0 Development Team* with Robert Lemke as the chairman was set up for the re-development.

The development team even considered a completely new system (e.g. including a change from Java to Ruby) but such significant changes were rejected in a brainstorming session unanimously by the attendees. An extensive re-factoring of the existing code was also refused so that the way forward became clear: the aim was to develop a totally new CMS based on PHP but maintaining important features such as TypoScript and the page tree.

The new system would also have to be capable of meeting new challenges and supporting new technologies such as SOAP, REST, WCAG, LDAP, XHTML, etc.

The following goals (development principles) were put forward:

- Iterative development rather than “big jumps”
- Small steps and frequent releases
- “Clean code” for fundamental changes
- “Unclean code” for working prototypes in order to reach a result quickly
- 100% innovating and 0% backwards compatibility
- Extract the “soul” of TYPO3 in order to keep it in the new product
- Code name “Phoenix”
- Architecture based on PHP 5.0

In addition, the decision was made for the first time that the development of the new CMS would be funded by the TYPO3 Association.[2] The core developers (Robert Lemke at day one, later Karsten Dambekalns as well) were employed by the Association full-time. This approach – not typical for an open source project – was justified by the fact that the new product should be released to market as soon as possible.

At the end of 2013, the development of TYPO3 Phoenix (aka “TYPO3 Flow” or “TYPO3 Neos”) consumed nearly 1 million Euro[3].

Shortly after the development started, there was confusion among the community and with clients, who considered using the system in the future and were not sure if it was worth learning TYPO3 version 4.x or waiting for TYPO3 version 5.x. There was also the question of whether or not someone would invest in a large TYPO3 version 4.x project, given the fact that a new major version will possibly become available *soon*.

1.2. The Berlin-Manifesto

To address these reasonable doubts, the so-called “Berlin-Manifesto”[4] has been published in October 2008. High-ranking representatives of the TYPO3 community gathered to work out a roadmap for TYPO3, both the TYPO3 version 4.x and the TYPO3 version 5.x. In the manifesto they stated:

- TYPO3 version 4.x continues to be actively developed
- TYPO3 version 4.x development will continue after the release of TYPO3 version 5.x
- Future releases of TYPO3 version 4.x will see its features converge with those in TYPO3 version 5.x
- TYPO3 version 5.x will be the successor to TYPO3 version 4.x
- Migration of content from TYPO3 version 4.x to TYPO3 version 5.x will be possible
- TYPO3 version 5.x will introduce many new concepts and ideas and there will be support and adequate resources to ensure a smooth transition

The signees of this statement were: Patrick Broens, Karsten Dambekalns, Dmitry Dulepov, Andreas Förthner, Oliver Hader, Martin Herr, Christian Jul Jensen, Thorsten Kahler, Steffen Kamper, Christian Kuhn, Sebastian Kurfürst, Martin Kutschker, Robert Lemke, Tobias Liebig, Benjamin Mack, Peter Niederlag, Jochen Rau, Ingo Renner, Ingmar Schlecht, Jeff Segars, Michael Stucki and Bastian Waidelich.

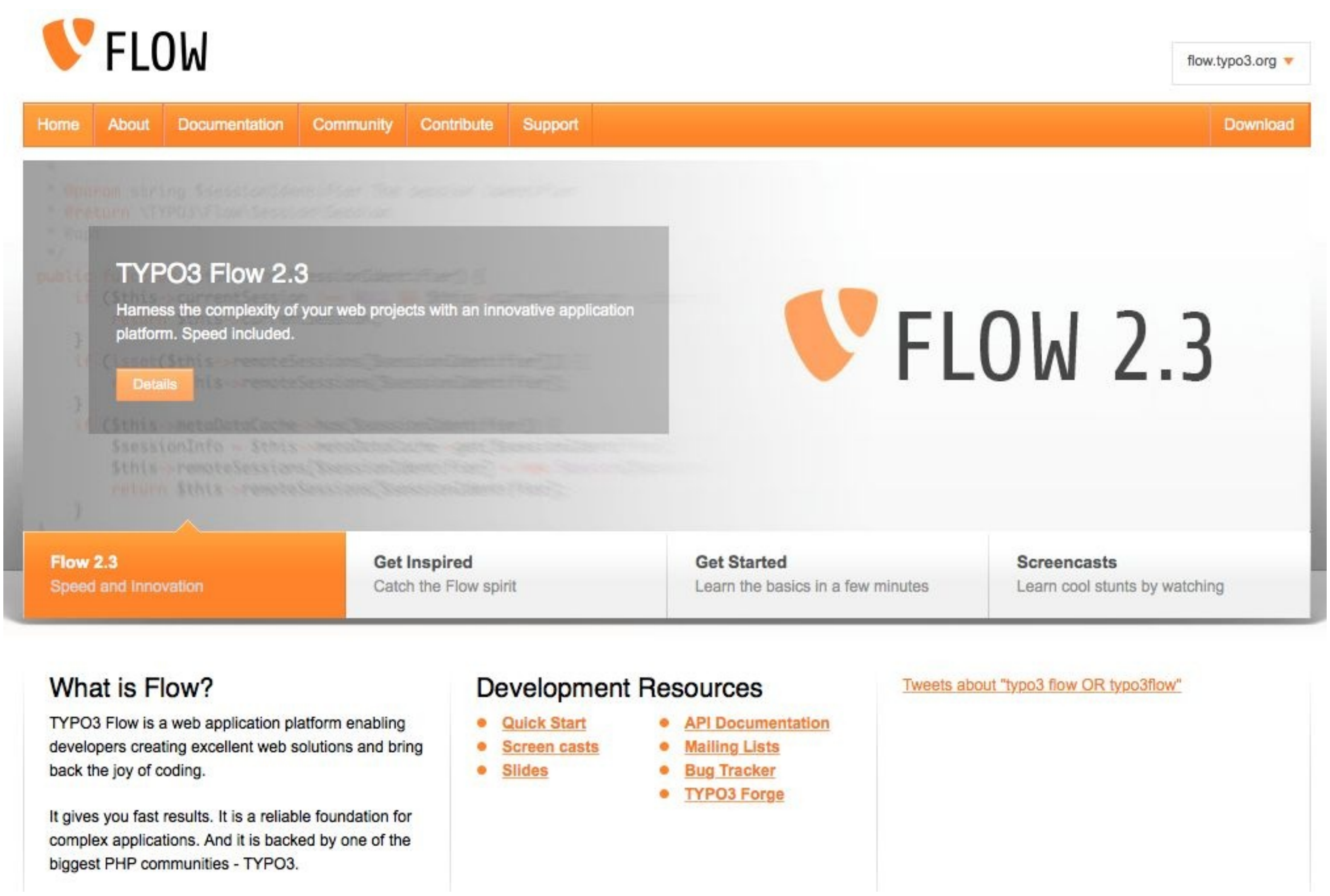
Figure 1.1. Signees of the Berlin-Manifesto in October 2006

After this pledge, the sceptics were quietened however it also became apparent that the new system would not be available for another 1 to 2 years. Unfortunately, the TYPO3 version 5.x Development Team did not have the courage or the vision to create a reliable roadmap, possibly due to concerns about what could happen if the goals were not met.

1.3. The Hour of Birth of TYPO3 Flow (FLOW3)

It soon became obvious that the new CMS required new components which were also useful outside of a CMS: MVC, templating, session handling, etc. This led to the conclusion that a framework should be developed first and based on that, the new CMS can grow.

Figure 1.2. The TYPO3 Flow website



TYPO3 Flow 2.3

Harness the complexity of your web projects with an innovative application platform. Speed included.

Flow 2.3
Speed and Innovation

Get Inspired
Catch the Flow spirit

Get Started
Learn the basics in a few minutes

Screencasts
Learn cool stunts by watching

What is Flow?

TYPO3 Flow is a web application platform enabling developers creating excellent web solutions and bring back the joy of coding.

It gives you fast results. It is a reliable foundation for complex applications. And it is backed by one of the biggest PHP communities - TYPO3.

Development Resources

- Quick Start
- Screen casts
- Slides
- API Documentation
- Mailing Lists
- Bug Tracker
- TYPO3 Forge

This marked the birth of FLOW3, which saw the light of day in June 2009 and today (early 2015) version 2.3 is available. It took more than two years to reach a final version and if you review the feature list carefully, you will realise that no version of FLOW3 prior 1.2.x or even 2.0.x deserve to be classified as “final”.

As part of a re-branding process of all products in the TYPO3 universe, FLOW3 was renamed *TYPO3 Flow* in October 2012.

Today, TYPO3 Flow unites a number of contemporary software design principles in its entirety as a framework – strictly following the paradigm “convention over configuration”. These principles are for example:

- Domain Driven Design (DDD)
- Model View Controller (MVC)
- Aspect Oriented Programming (AOP)
- Dependency Injection (DI)

This speeds up the development process significantly because a lot of useful assumptions have already been made for the developer. This means they can focus on their specific challenge and concentrate on finding its solution.

In general terms, TYPO3 Flow can be understood as an application framework or actually “TYPO3 Flow Enterprise PHP Framework” as the TYPO3 Flow Team calls it. Therefore, it can be used independently of TYPO3 CMS for any web application. It is possible to use TYPO3 Flow to develop a webservice, a CRM, an online shop or a simple website – the latter with a self-written content management of course.

Among competitors such as Symfony, Zend Framework, Cake and many others, TYPO3 Flow counts as amongst the most modern, advanced and technically mature PHP frameworks on the market. Its inventors pick the features, technologies and methodologies, which they integrate with great care and implement only the best.

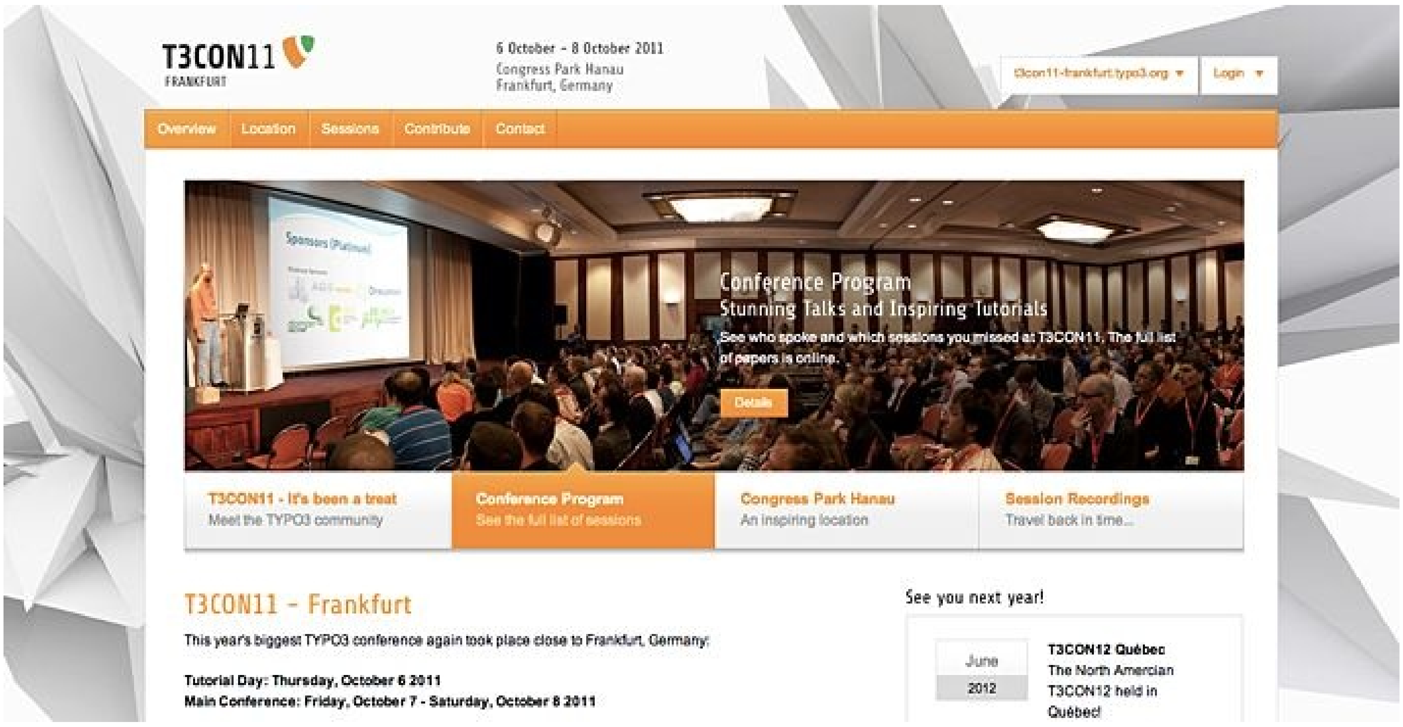
1.4. Like Phoenix from the Ashes

The decision to initially focus on TYPO3 Flow seemed reasonable but there was again growing uncertainty over when the eagerly awaited TYPO3 version 5.x would be published.

The first Phoenix release was eventually published as “Phoenix Sprint Release 1” in June 2010 – and from then on, further spring releases up to number 8. These were far away from being an alpha, beta or even a final version and this did not go unnoticed by the TYPO3 community, who expected a much better result after more than five years.

Yielding under growing pressure, the TYPO3 version 5.x Team announced that the website for the TYPO3 Conference 2011[5] (T3CON11 held in Frankfurt/Germany) would be the first site built with TYPO3 version 5.0 – accompanied by a huge marketing stunt.

Figure 1.3. The first website built with TYPO3 5.0



Conference Program
Stunning Talks and Inspiring Tutorials
See who spoke and which sessions you missed at T3CON11. The full list of papers is online.

T3CON11 - Frankfurt

This year's biggest TYPO3 conference again took place close to Frankfurt, Germany.

Tutorial Day: Thursday, October 6 2011
Main Conference: Friday, October 7 - Saturday, October 8 2011

T3CON12 Québec
The North Amercian T3CON12 held in Québec!

Only insiders realised that the website did not feature any “backend capabilities” (in fact a XML file was used to build the site rather than real content management functions). However members of the community and especially clients drew hope from these visible results. They assumed, that version 5.0 was close to being published, given the fact that it was already possible to build a website such as the one for the TYPO3 Conference.

The bottom line was, that a system existed, which showcased itself as almost complete but its real status can only be classified as a pre-alpha status. At the same time, the version number of the 4.x branch grew steadily and in October 2011 version 4.6 was released.

Assuming that the release cycle continues at this pace, TYPO3 version 5.x must be delivered in about 1.5 years in order to ensure the version number of 4.x does not come too close to 5.x.

However experience shows that 6 years of development time is insufficient to finish a stable system ready for production use. How could it even be possible to deliver a system in 1.5 years from now, that is the successor of TYPO3 version 4.x according to the Berlin-Manifesto?

1.5. The TYPO3 Dilemma

The TYPO3 Core Team acted quickly and decided in the last quarter of 2011, not to follow its original plans. They escaped the dilemma by terminating the plan of TYPO3 version 5.x and officially announced the decision during the T3BOARD12 – somewhat inappropriately.[6]

- A version 5.x will never exist.
- The next TYPO3 version 4.x will not be named TYPO3 4.8 (as the successor of TYPO3 4.7), but will be TYPO3 version 6.0 (published in October 2012).
- The current TYPO3 version 5 will be renamed to TYPO3 Phoenix 1.0 temporarily and will receive a new name over the year 2012 (“TYPO3 Neos”).
- TYPO3 Phoenix is not the successor of TYPO3 4.x any more but a discrete product, with some similarities of TYPO3 4.x.
- TYPO3 4.x may (and will) allow for breaking changes.

This announcement resulted in a storm of protest, never seen before on this scale.[7] This was hardly surprising due to the fact that many people were short changed and could not follow the rationale. Additionally, comments from the public highlighted the disappointment, that even after 6 years of development time and a lot of money already spent on this project, there were no actual results.

However on a positive note, it was now possible to break free from the chains of the past and put all the energy into the new version, for example in breaking changes such as the File Abstraction Layer (FAL). This resulted in new motivation and drive and as an outcome TYPO3 version 6.0 became the best release of the TYPO3 history so far.

1.6. The New TYPO3 Universe

Therefore, the TYPO3 universe consists of three independent products:

- TYPO3 CMS x.y – current stable version 6.2 LTS (support guaranteed until 2017), 4.5 LTS (support reached its end-of-life in March 2015)[8]
- TYPO3 Flow 2.3
- TYPO3 Neos 1.3

Neos and TYPO3 CMS are in fact completely different content management systems, addressing different target audiences. They both belong to the same sort of CMS products and share their developers' wish to have a migration tool at one point in the future.

A concrete migration concept does not exist yet. However system extensions Extbase & Fluid have been integrated in TYPO3 4.x/6.x in 2009. This allows developers to build extensions, which are technically very similar to TYPO3 Flow packages. This aims to minimise the migration efforts.

This also means, that TYPO3 CMS will last for quite a long time and this is where this book comes into play: by using Extbase & Fluid, extensions will be developed, which may work in TYPO3 CMS exclusively – or may be ported to TYPO3 Neos at one point.

For a long time, developers pushed for a strict upward compatibility between Extbase and TYPO3 Flow but in March 2012 they realised that it was impossible to allow breaking changes in TYPO3 4.x/6.x and also aim for being in sync with TYPO3 Flow. It can be assumed, that every Extbase & Fluid code developed for TYPO3 up to version 4.7 can also be made executable in TYPO3 Flow and Phoenix with little effort. After then, it might be more complicated because of the risk that both concepts diverge. Although, the Extbase Team tried hard to keep the sync and at the ACME 2012 (*Active Contributor Meeting*) a decision was made, that Extbase and TYPO3 Flow should converge again.

1.7. The History of Extbase & Fluid

The architecture change to TYPO3 Flow (and subsequently TYPO3 Neos) had a significant consequence though, it is irrelevant for a core system if the entire source code changes but this is not the case for extensions.

Currently, more than 6000 extensions are available for TYPO3 CMS 4.x/6.x and none of them are usable in TYPO3 Neos. Ultimately, they all need to be rewritten from scratch – or at least the most important ones.

However this is not required immediately and the Berlin-Manifesto explicitly states that the development of TYPO3 4.x/6.x will continue but the experience shows that clients often prefer the latest technology for new projects so that they are future proof. After all, the development often consumes a lot of time and money and nobody wants to be forced to spend the budget again after a technology change.

It would be great if it would be possible to develop extensions today, using the latest methodologies and paradigm of TYPO3 Neos and to migrate them with a minimum investment of time and effort as soon as TYPO3 Neos becomes widely adopted.

1.7.1. Backports: Extbase & Fluid

Jochen Rau, who developed the *Extbase* extension (which is part of the TYPO3 system core since version 4.3), achieved exactly this. Extbase is a backport of the features in TYPO3 Flow, which are required to run extensions in TYPO3 as well as in TYPO3 Neos (or with TYPO3 Flow) with minimal changes required.

It is understandable that not all new concepts can be adopted – simply because these are completely different architectures but the most important have been implemented: first and foremost the MVC framework, which has been ported almost in its entirety. Also, basic methods of the Domain Driven Design (DDD) have been transferred and a new templating engine, too: *Fluid*.

And that is not all – you will see, there are many new and exciting things to discover in TYPO3 even today. The best is, the knowledge you will gain from this can also be applied for projects under TYPO3 Neos.

1.7.2. Programming on an Advanced Level

We will cover all these new concepts and methodologies in the following chapters in detail, but it should be pointed out, that the learning efforts required are not unremarkable, in particular the PHP knowledge of a software developer.

In TYPO3 it was relatively easy to develop extensions by using dirty, procedural code (by the use of `pi_base` for example). This is no longer possible with Extbase. Everything (and really everything) is based on objects now, which requires advanced knowledge and understanding of object-orientated development in PHP.

This is the perfect time, to deal with this subject and we can highly recommend the books “PHP This! A Beginners Guide to Learning Object Oriented PHP” by Michelle Gosney[9] and “Learning PHP Design Patterns” by William Sanders.[10]

To ensure, we are on the same knowledge level, the next chapter summarises all concepts and methodologies required for programming in Extbase. If you already have adequate experience in object-orientated programming, you can skip this chapter confidently.

---

[1] http://www.slideshare.net/robertlemke/t3dd06-typo3-50-brainstorming-results

[2] http://association.typo3.org

[3] cf. “Financial Statements” of the TYPO3 Association, e.g. http://association.typo3.org/fileadmin/documents/financial_statements/Typo3_Abschluss_2008_Rev_D.pdf and the following years.

[4] http://typo3.org/roadmap/berlin-manifesto/

[5] http://association.typo3.org/home/news/news-detail/news/first_typo3_50_website_launched////ref/assoc/

[6] http://buzz.typo3.org/people/xavier-perseguers/article/typo3-60-at-the-corner-how-is-it-possible/

[7] http://lists.typo3.org/pipermail/typo3-english/2012-March/thread.html#79513

[8] TYPO3 version 4.7, 6.0 and 6.1 have been discontinued already.

[9] http://www.phpthis.com

[10] http://www.oreilly.de/catalog/9781449344917/

Chapter 2. The Basics of Object-Orientated Programming

As the name indicates, *objects* are the focus of object-orientated programming. This concept is easier to grasp if you put computer programming aside and think about objects in our daily lives.

We are surrounded by objects in everyday life; cars, smartphones, computers, cocktails, food, factories, televisions, etc. All these objects feature specific attributes such as colours, smell, look, alcoholic strength or distance driven. In addition, all objects have some functions; a computer can be switched on, a car can be driven and a smartphone allows us to send text messages as well as using the Internet. These properties and functions are closely connected with the objects and can be considered as one unit.

OOP in Extbase & Fluid

Extbase & Fluid does not make use of all features of PHP’s object-orientated programming. For example, there are no final classes nor the *private* visibility. Therefore, this chapter does not cover elements, which are not used by Extbase & Fluid.

2.1. Classes and Objects

If you compare two cars for example, it is obvious that they have a lot in common – they both have wheels, an engine and a steering wheel. Therefore it is possible to introduce an abstraction layer and call the abstract car a *class*.

This is a sort of construction plan (or blue-print) for a concrete car object. A class also shows properties and functions but these are not concrete. A class “car” features an “engine” but the concrete type (e.g. “N54B30”) will not be implemented until the object has been derived later. This design is called *instantiation* or *derivation* – an object is an *instance of a class*.

In the following, we will develop a program that manages cars. Firstly, we have a class “Car”, which has the attributes “producer”, “colour” and “mileage”. In the OOP universe, these attributes are called *properties*.

The PHP code for this example looks like the following:

```
class Car {
   public $producer;
   public $colour;
   public $milage;
}
```

The keyword `class` starts the class, followed by a pair of curly brackets. Also note the syntax: the opening bracket and the class name is separated by a space, whereas the closing bracket stands in its own line at the beginning.

TYPO3 Coding Guidelines

Many coding syntax in TYPO3 are defined and specified in the official TYPO3 Coding Guidelines.[11]

The properties are listed line-by-line inside the class. Every property has the keyword `public` prefixed (we will explain this a little bit later), followed by the `$` character and the name. This list is also called class *declaration*.

Syntax

In Extbase, class names always start with an upper case letter, followed by lower case letters as long as a new unit of meaning starts, then an uppercase letter – for example: `ThisIsAClassName`. This notation is called

UpperCamelCase. Properties (and later on methods) start with a lower case letter, e.g. `thisIsAPropertyName`. This notation is called *lowerCamelCase*.

2.2. Methods

Now we would like to provide our car with some “functions” – these are called *methods* in object-orientated programming. We will implement a few methods so we can start and stop the engine as well as driving the car.

```
class Car {

   public $producer;
   public $colour;
   public $milage;

   public function startMotor() {
   }

   public function drive($kilometer) {
   }

   public function stopMotor() {
   }

}
```

A method always consists of two parts:

Method signatur
: It starts with the keyword `public`, then `function` and after that the name in lowerCamelCase notation followed by a pair of round brackets. Inside these brackets, you can possibly find parameters (if applicable) which are passed to the method.

Method body
: The body of a method is either a block wrapped between opening and closing brackets, or just a semicolon (this is the case with abstract methods). All functionality of a method can be found in the body.

As a result, our class as a construction plan for a concrete object is complete and can be used to build arbitrary objects, which feature the defined properties and methods.

```
$audi = new Car();
$bmw = new Car();
```

The derivation or instantiation happens by the operator `new`. We have built an Audi and a BMW and can work with them directly.

The PHP function `print_r()` allows us to output the object at any time.

```
print_r($audi);
```

```
// Output:

Car Object ( [producer] => [colour] => [milage] => )
```

The PHP function `var_dump()` even prints more details about the object:

```
var_dump($audi);

// Output:

object(Car)#1 (3) {
  ["producer"]=>
  NULL
  ["colour"]=>
  NULL
  ["milage"]=>
  NULL
}
```

2.2.1. The Arrow Operator

In order to access the properties and methods of a class, we use the *arrow operator*, which consists of a dash and a greater-than sign: `->` We have read and write access by using this operator.

```
...
$audi->producer = 'Audi';
$audi->startMotor();
echo 'The manufacturer of the car is: ' . $audi->producer;
```

2.2.2. The Constructor

When a factory builds a vehicle, the car already features a manufacturer, a colour and a mileage (which will start at 0). Therefore it would make sense to set these properties as soon as the class gets instantiated.

To do so, we use a *constructor* – a specific function, which will be called automatically as soon as the object is being derived from the class.

The name of a constructor is always `__construct` (with two underscores) and there can only be one constructor per class.

```
class Car {
...
   public function __construct($producer, $colour, $milage = 0) {
      $this->producer = $producer;
      $this->colour = $colour;
      $this->milage = $milage;
   }
...
}
```

Constructors always appear as the first method of a class for clarity reasons, even if in theory the position is arbitrary.

In order to instantiate an object, we could use for example:

```
$bmw = new Car('BMW', 'red');
$audi = new Car('Audi', 'black', 200);
```

The two (respectively three) parameters are passed to the constructor method automatically. The constructor excepts three parameters, the last one is optional (a default value has been set in the method signature). Without the third parameter, this default value (in this case `0` kilometres) is used. All other parameters must exist when the object is created, otherwise an error message `Missing argument` occurs.

2.2.3. Access by Using $this

Parameters passed to the constructor are used to set the properties of the class. However there is a dilemma here, the name of the object is unknown at this point in time and therefore we are not able to use it. The operator `$this` addresses this issue – it always refers to the current instance.

2.2.4. Filling Methods with Content

At this point, a factory is able to produce cars, which should be tested before delivery. We will add some functionality to the methods, which should achieve the following:

1. start the engine
2. drive 10 kilometres
3. stop the engine

In order to ensure this works reliably, a new property `$isEngineStarted` is needed to distinguish between various states of the engine.

```
class Car {

   public $producer;
   public $colour;
   public $milage;

   public $isMotorStarted = FALSE;

   public function __construct($producer, $colour, $milage = 0){
      $this->producer = $producer;
      $this->colour = $colour;
      $this->milage = $milage;
   }

   public function startMotor() {
      if ($this->isMotorStarted === FALSE) {
         $this->isMotorStarted = TRUE;
      }
   }

   public function drive($kilometer) {
      if ($this->isMotorStarted === TRUE) {
         $this->milage = $this->milage + $kilometer;
      }
   }

   public function stopMotor() {
      if ($this->isMotorStarted === TRUE) {
         $this->isMotorStarted = FALSE;
      }
   }

}
$bmw = new Car('BMW','red');
$bmw->startMotor();
$bmw->drive(10);
$bmw->stopMotor();

var_dump($bmw);
```

The result looks like the following:

```
object(Car)#1 (4) {
  ["producer"]=>
  string(3) "BMW"
  ["colour"]=>
  string(3) "red"
  ["milage"]=>
  int(10)
  ["isMotorStarted"]=>
  bool(false)
}
```

2.3. Inheritance of Classes

The functionality of the class we just created is quite presentable already. But what if we want to manage a different type of car with it? A Cabriolet for example has different (respectively additional) attributes, a convertible top for instance, which can be open or closed.

There would be no reason to create a completely new class for this but we could re-use the existing class as a basis and expand on it.

This can be achieved by using the keyword `extends`. With this keyword, the new class inherits all properties and methods of the parent class and can be modified and/or extended. This is called *inheritance* or *expansion*.

Firstly, all properties and methods are available in the inherited class unaltered and can be overwritten (by implementing a function of the parent class again) or extended.

```
class Cabriolet extends Car {
   public $convertibleTopOpen = FALSE;

   public function openConvertibleTop() {
      if ($this->convertibleTopOpen == FALSE) {
         $this->convertibleTopOpen = TRUE;
      }
   }

   public function closeConvertibleTop() {
      if ($this->convertibleTopOpen == TRUE) {
         $this->convertibleTopOpen = FALSE;
      }
   }
}

$bmw = new Cabriolet('BMW', 'red');
$bmw->openConvertibleTop();

print_r($bmw);
```

The result:

```
Cabriolet Object
(
    [convertibleTopOpen] => 1
    [producer] => BMW
    [colour] => red
    [milage] => 0
    [motorStarted] =>
)
```

2.3.1. Access by Using parent

The following code ensures, the convertible top closes, before the engine is stopped (if the top is currently open):

```
class Cabriolet extends Car {
...
   public function stopMotor() {
      if ($this->convertibleTopOpen == TRUE) {
         $this->convertibleTopOpen = FALSE;
      }
      if ($this->motorStarted == TRUE) {
         $this->motorStarted = FALSE;
      }
   }
   ...
}

$bmw = new Cabriolet('BMW', 'red');
$bmw->startMotor();
$bmw->openConvertibleTop();
$bmw->stopMotor();
```

This works as expected but the approach has a downside: the method `stopMotor()` has been implemented twice. First for the class `Car` and second for the class `Cabriolet`. If something changes in class `Car`, we have to adjust this change in the second class too.

In an ideal world we would be able to access the method of the parent class directly and this is possible by using the keyword `parent`. The function call is static (no new object of the parent class is being created), thus we can not use the arrow operator. Static calls are executed by the double colon operator `::` (also called *Scope Resolution Operator*).

```
class Cabriolet extends Car {
...
   public function stopMotor() {
      if ($this->convertibleTopOpen == TRUE) {
         $this->convertibleTopOpen = FALSE;
      }
      parent::stopMotor();
   }
...
}
```

2.3.2. Verifying Class Derivation

It can be useful to determine from which parent class an existing class has been derived. This is done by the `instanceof` operator:

```
$bmw = new Cabriolet('BMW', 'red');

if ($bmw instanceof Car) {
    echo "The car is of type Car";
}

if ($bmw instanceof Cabriolet) {
    echo "The car is of type Cabriolet";
}

// Output
The car is of type Car
The car is of type Cabriolet
```

2.4. Abstract Classes

Usually every class contains the entire code which is required for later use. However sometimes the code a class will contain is unknown at the time the class is implemented.

For example, a route guidance system is not mounted into the car while the vehicle is still in the factory. The customer chooses the device at the car dealer and then has it installed.

Assuming that every car will have a navigation device, the concrete implementation will be postponed to "later". However we want to ensure that the methods are always consistent, e.g. to be able to turn on or turn off the device, independent of its concrete characteristics.

For these cases, we use *abstract* classes:

```
class Car {
...
   abstract public function startNavigationDevice();
   abstract public function stopNavigationDevice();
...
}
```

A class is `abstract`, if at least one of its methods is `abstract`. When this is the case, the keyword `abstract` should be placed in front of the class name. The methods are abstract because all their implementations are missing – they do not feature any method bodies.

At this point, the class can not be used any more. As soon as you try to create an instance of a Cabriolet, you get an error message about the class `Car` immediately:

```
Fatal error:  Class Car contains 1 abstract method and must therefore be dec
```

If you enter the keyword `abstract` in front of the class `Car`, you get another error – this time in relation to the class `Cabriolet`:

```
Fatal error:  Class Cabriolet contains 1 abstract method and must therefore
```

Our only option left is to *concretise* the abstract methods by adding a method body, even if it remains empty. We will do this in class `Cabriolet` as follows:

```
class Cabriolet {
...
   public function startNavigationDevice() {

   }
   public function stopNavigationDevice() {

   }
...
}
```

The purpose of abstract classes is their implementation is postponed but we already know their method names and method signatures when we create the class and we can work with them.

2.5. Interfaces

The methods which a class must consist of can be defined by an *Interface*. In this connection only method signatures are implemented and method bodies and especially properties are not part of the Interface definition. Therefore, an Interface of a class is similar to a class, which only consists of abstract methods. We can not use properties but constants.

We sill store the guidance device in such an Interface to separate the code and also to use the Interface in other car types later, without writing redundant code. Despite the fact that methods of an Interface are abstract, the keyword `abstract` must not be used.

```
interface NavigationDevice {
   public function startNavigationDevice();
   public function stopNavigationDevice();
}
```

An Interface always starts with the keyword `interface`. The terminology "implementation of an Interface" is used, rather than "derivation of an Interface" as explained before. The obligation to concretise every method later remains. In addition, multiple Interfaces can be applied at the same time (Interface names separated by commas).

```
class Car implements NavigationDevice {
   ...
}

class Car2 implements NavigationDevice, \TYPO3\CMS\Core\SingletonInterface {
   ...
}
```

The bottom line is we can `implement` multiple classes at once but we can only derive (`extend`) from one class.

2.6. Visibility: Public and Protected

We can define, how to access methods and properties from the "outside". This means both the main program and another class, instance or object.

All methods and properties have been declared as *public* so far. This allows access to the method or property by another class as well as by the main program.

If the keyword `protected` is used instead of `public`, only the class to which the method or property belongs and its derived methods and properties can access it.

The main program does not have this access though. This circumstance is called *encapsulation* of data.

```
class Car {
   ...
   protected $colour;
   ...
}
```

If you try to access the property `$colour` from the outside, an error occurs:

```
...
$bmw = new Cabriolet('BMW', 'blue');
$bmw->colour = 'red';

// Result:
Fatal error:  Cannot access protected property Cabriolet::$colour
```

It's not possible to access the colour from the outside, which makes sense because the colour of a car can not be changed easily. However you can drive to a garage and get your car painted a different colour so there must be a way to access it somehow.

Visibilities in Extbase & Fluid

All properties in Extbase & Fluid are `protected` and all methods are `public` (except really internal methods).

2.6.1. Getter and Setter

We just learnt that visibilities aim to control the access to the code. In order to allow access to `protected` properties, we have to develop specific methods, which will allow this. These methods are called *getter* (get, because they allow read access) and *setter* (set, because they allow write access). As per convention, the names of these methods start with `get`, respectively `set` and the property names are stated in UpperCamelCase.

```
class Car {
   ...
   protected $colour;
   ...
   public function getColour() {
      return $this->colour;
   }

   public function setColour($colour) {
      $this->colour = $colour;
   }
   ...
}

$bmw = new Cabriolet('BMW', 'blue');
$bmw->setColour('red');
```

2.7. Type Hints

The source code can be reduced even further by using *Type Hints*. As an example, the software should provide a method `rent()` to allow hiring the car. The method should ensure, that the vehicle is really a car.

```
public function rent($vehicle) {
   if ($vehicle instanceof Car) {
      // We made sure, the car can be hired now
   }
}
```

It's possible to cut this short by using a type hint. The type of the parameter is defined by adding it in front of the variable, separated by a space. This can be used with class names as well as interfaces.

```
public function rent(Car $vehicle) {
   ...
}
```

Datatypes `array` and class names are valid type hints but standard types such as `integer`, `string` or `boolean` are not.

There is another difference between both methods, the latter shows an error message if the call of the method violates the type hint:

```
Catchable fatal error: Argument 1 passed to Book\Extbase\Code\Cabriolet::ren
```

2.8. Static Calls

All examples instantiated objects from classes so far. This is reasonable because we do not know how the object will behave exactly.

If the object's behaviour is the same every time, it would be unwise to instantiate an object each time. It would be convenient to have a static access to the class.

Consider a price list as an example, we can expect the same result for every input parameter, this is classic example of a static call:

```
class PriceList {

   const DATE = '2013';

   static public function getPrice(Car $car) {
      if ($car->producer == 'BMW') {
         return '100 EUR';
      }
      if ($car->producer == 'Audi') {
         return '95 EUR';
      }
   }
}

$bmw = new Cabriolet('BMW', 'red');
echo 'Price list of: ' . PriceList::DATE . '<br>';
echo 'The car costs: ' . PriceList::getPrice($bmw) . ' per day!';

// Result
Price list of: 2013
The car costs: 100 EUR per day!
```

We are required to add the keyword `static` to all functions.

2.9. Namespaces

Extbase uses namespaces since version TYPO3 CMS 6.0 – just as the CMS itself.

Namespaces are useful when there is the risk that two classes could have the same name. In order to tackle this risk, long and complicated class names were used in Extbase prior version 4.7, e.g. `Tx_Extbase_Validation_Validator_AbstractValidator`. At the time, the class name indicated where to find the file of the class in the system (we will come back to this later).

A namespace is set by the keyword `namespace`.

```
namespace Book\Extbase\Code;

$bmw = new Car('BMW', 'red');
print_r($bmw);

// Results in the following output:

Book\Extbase\Code\Car Object
(
    [producer] => BMW
    [colour] => red
    [milage] => 0
    [motorStarted] =>
)
```

Whenever a class is used, the namespace will be “added” automatically, which results in a *fully qualified class name* (FQCN).

PHP relinquishes this addition, when the class name starts with a backslash `\` (this labels the class name as “absolute”). In this case, PHP assumes that the class name is already fully qualified.

In order to cut another (fully qualified) name short, we can use the keyword `use` as follows:

```
namespace Book\Extbase\Code;

use Vehicle\Motor\FourWheels;

class Car extends FourWheels {
   ...
}
```

In this case, the fully qualified name `Vehicle\Motor\FourWheel` will be used instead of the namespace. Therefore, `use` acts as a kind of alias. The last element of the namespace referenced by `use` is used as the name (here: `FourWheel`), unless a different name has been specified by using the keyword `as`:

```
namespace Book\Extbase\Code;

use Vehicle\Motor\FourWheels as FW;

class Car extends FW {
   ...
}
```

2.10. Important Design Patterns

In software development, you will face design challenges sooner or later. Solutions for specific challenges are often very similar and *design patterns* have been established in order to tackle some of these.

More than 80 different design patterns have proven themselves but only a fraction of those are used in modern software design today.

The following three design patterns are often used in Extbase & Fluid, which we will explain in the following sections in more detail:

- Singleton
- Prototype
- Dependency Injection

2.10.1. Singleton

If we use the car manufacturer example discussed earlier and add an employee who counts the completed vehicles at the end of the production line (in terms of software development, this employee is an “object”). One employee (or object) would be required for each conveyor and assuming, the number of those objects is known, we can ask each one of them for the number of vehicles counted and sum these figures up in order to get the total number of completed vehicles.

However it would be much easier, if – when such as an object is created – a check would be executed, if an object of this class already exists and if it does, this would be used instead of creating a new one. Otherwise, a new object is being created. This ensures, that only one instance of an object exists at runtime.

This would result in more work for the employee because they have to hop from one conveyor to the next – but we only need to ask them for the total number of completed vehicles at the end of the day.

In Extbase a class of type Singleton has to implement the interface `\TYPO3\CMS\Core\SingletonInterface` as follows:

```
class Object implements \TYPO3\CMS\Core\SingletonInterface {

}
```

The interface remains empty:

```
namespace TYPO3\CMS\Core;

interface SingletonInterface {

}
```

In order to get an instance of class `object`, a specific function such as `makeInstance()` must be used rather than `new`:

```
$obj = \TYPO3\CMS\Core\Utility\GeneralUtility::makeInstance('Object');
```

This method checks whether the object matches the design pattern `Singleton`. In this case, the method uses a kind of register to determine, if an instance of the class already exists and return this. If no instance exists yet, a new class is instantiated (by using `instantiateClass()`) and the instance returned.

The method `makeInstance()` is defined as follows:

```
static public function makeInstance($className) {
      if (!is_string($className) || empty($className)) {
         throw new \InvalidArgumentException('$className must be a non empty
      }
      $finalClassName = self::getClassName($className);
      // Return singleton instance if it is already registered
      if (isset(self::$singletonInstances[$finalClassName])) {
         return self::$singletonInstances[$finalClassName];
      }
      // Return instance if it has been injected by addInstance()
      if (
         isset(self::$nonSingletonInstances[$finalClassName])
         && !empty(self::$nonSingletonInstances[$finalClassName])
      ) {
         return array_shift(self::$nonSingletonInstances[$finalClassName]);
      }
      // Create new instance and call constructor with parameters
      $instance = static::instantiateClass($finalClassName, func_get_args())
      // Create alias if not present
      $alias = \TYPO3\CMS\Core\Core\ClassLoader::getAliasForClassName($final
      if ($finalClassName !== $alias && !class_exists($alias, FALSE)) {
         class_alias($finalClassName, $alias);
      }
      // Register new singleton instance
      if ($instance instanceof \TYPO3\CMS\Core\SingletonInterface) {
         self::$singletonInstances[$finalClassName] = $instance;
      }
      return $instance;
   }

   protected static function instantiateClass($className, $arguments) {
      switch (count($arguments)) {
         case 1:
            $instance = new $className();
            break;
      ...
```

Extbase uses a special method instead of `makeInstance()` because *Dependency Injection* (see below) is used in addition to the Singleton concept.

2.10.2. Prototype

Prototype is the counterpart of Singleton to a certain extent and ensures, that a new instance is always created, whenever a new object is requested. Extbase treats a class as a prototype by default, if it has not been “marked” as a SingletonInterface.

2.10.3. Dependency Injection

Dependency Injection (DI) is a software design pattern which aims to simplify the resolution of dependencies of an object. It uses inversion of control in order to free an object from unnecessary connections, which are only required to resolve its dependencies but not for its main purpose. The responsibility of resolving the dependencies are transferred from the object to the framework (here: Extbase).

A typical example in Extbase would be a *repository* in a controller, which is often required. If you would instantiate the repository manually every time and change its implementation at one point in the future, you would have to review your entire code and update it.

Dependency Injection uses the opposite approach by telling the framework, which class you need and the framework searches for an appropriate implementation and “returns” this. The benefit is that you could for example return a different implementation globally by simply updating the configuration rather than each line of code, which refers to the implementation.

There are two types of DI in Extbase: *Constructor DI* and *Setter DI*. With the latter, you can choose between using the annotation `@inject` (see Section 2.11) and using your own injection method (which must start with `inject`).

The first option is recommended and looks like this:

```
/**
 * blogRepository
 *
 * @var \Lobacher\Simpleblog\Domain\Repository\BlogRepository
 * @inject
 */
protected $blogRepository;
```

The “inject method” could look like the following example:

```
/**
 * blogRepository
 *
 * @var \Lobacher\Simpleblog\Domain\Repository\BlogRepository
 */
protected $blogRepository;

public function injectBlogRepository(\Lobacher\Simpleblog\Domain\Repository\
    $this->blogRepository = $blogRepository;
}
```

The “Constructor Dependency Injection” could look like:

```
/**
 * blogRepository
 *
 * @var \Lobacher\Simpleblog\Domain\Repository\BlogRepository
 */
protected $blogRepository;

public function __construct(\Lobacher\Simpleblog\Domain\Repository\BlogRepos
   $this->blogRepository = $blogRepository;
}
```

The last example is somewhat tricky because you would usually get an error message due to the fact, that no parameters are passed to the Constructor when the object is being created. However Extbase takes care of generating the correct parameter and passes it to the object automatically based on the annotation and the type hint given.

2.11. Annotations

Annotations are meta information, which are used for methods, properties and classes. They have their own syntax and are parsed by a special Reflection-API. At this juncture, the code (in particular the annotations) are analysed and new, executable code is generated, which reacts on the logic stated in the annotations. This implements additional functionality to the code.

Annotations are part of the comments, which always start with `/**`. The annotation itself starts with the `@`-character:

```
/**
 * blogRepository
 *
 * @var \Lobacher\Simpleblog\Domain\Repository\BlogRepository
 * @inject
 */
protected $blogRepository;
```

In this example, a Blog repository is determined and assigned to variable `$blogRepository` by using Dependency Injection. In order to achieve this, the code will be reflected (parsed and analysed), written to a new location (e.g. cache) as required and then executed.

Thus annotations are essential for the sequence of the program and must not be left out or removed by PHP or any other software or library such as an PHP accelerator. See chapter Chapter 4 for further details.

---

[11] http://docs.typo3.org/TYPO3/CodingGuidelinesReference/

Chapter 3. Domain Driven Design

The design and development of software is always a creative process, where you are looking for smart solutions to tackle challenges.

The first action in this process should be a clear definition of the requirements. If this is incomplete or incorrect, misunderstandings occur, which inevitably result in a situation where the software does not meet all of the client’s expectations.

Therefore, it is essential to work closely with your client and fully understand their “problems”. In the domain driven design concept, the subject area (the sphere of your client’s business activities) is called *domain*.

The domain stands in the centre of the software design. The term *Domain Driven Design* was coined by Eric Evans in his book “Domain Driven Design: Tackling Complexity in the Heart of Software”, which is already 10 years old.

A different understanding of domain-specific concepts seems to be the main reason for discordances between user (clients) and application developer (service provider).

DDD (Domain Driven Design) is based on two important assumptions:

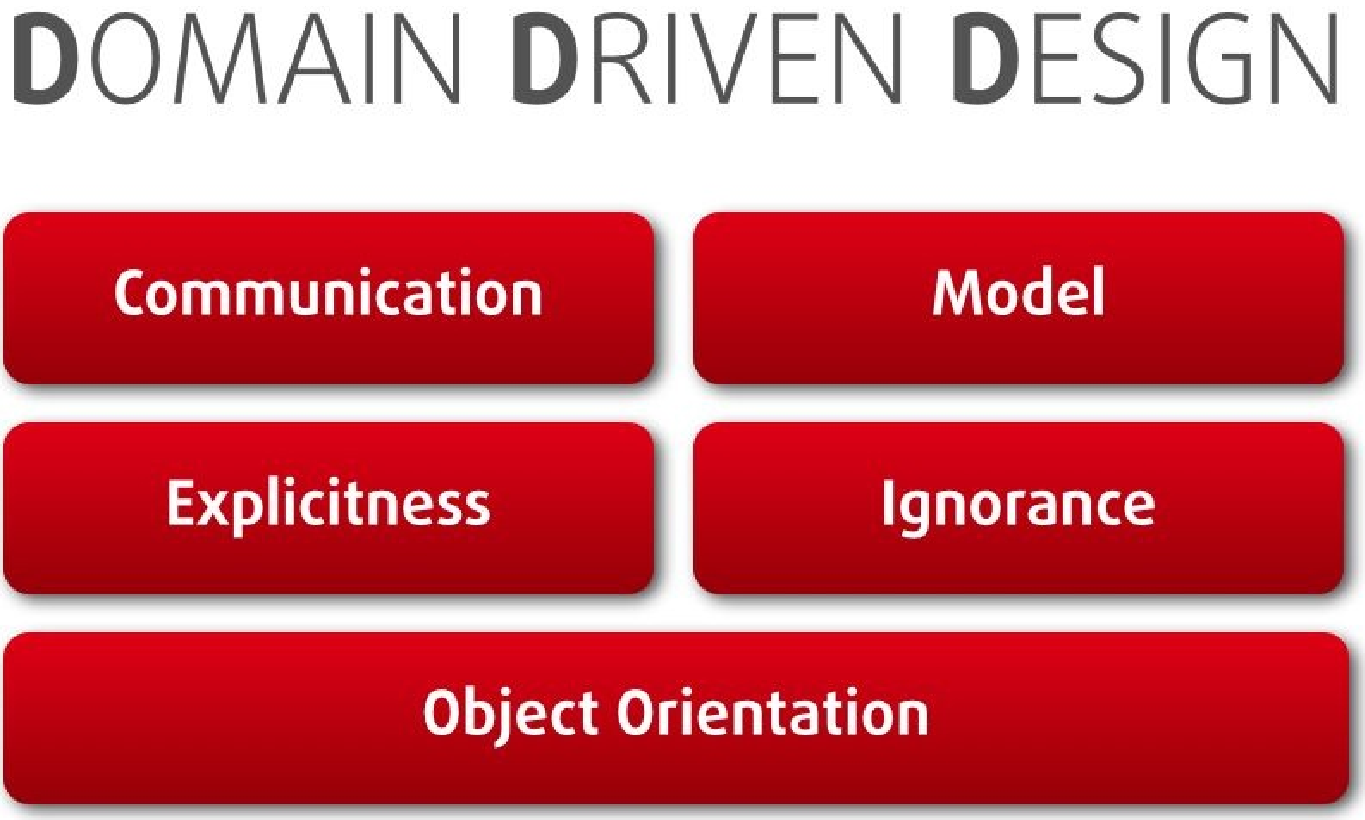
- The software design’s main focus should be on the core domain and domain logic.
- Complex designs should be based on a model of the domain.

This ensures that implicit relations (which often cause communication problems) become explicit.

3.1. Infrastructure Ignorance

DDD concentrates on the client’s domain, whereby other aspects can take a back seat or can be ignored, for example the infrastructure.

Figure 3.1. The concept of Domain Driven Design

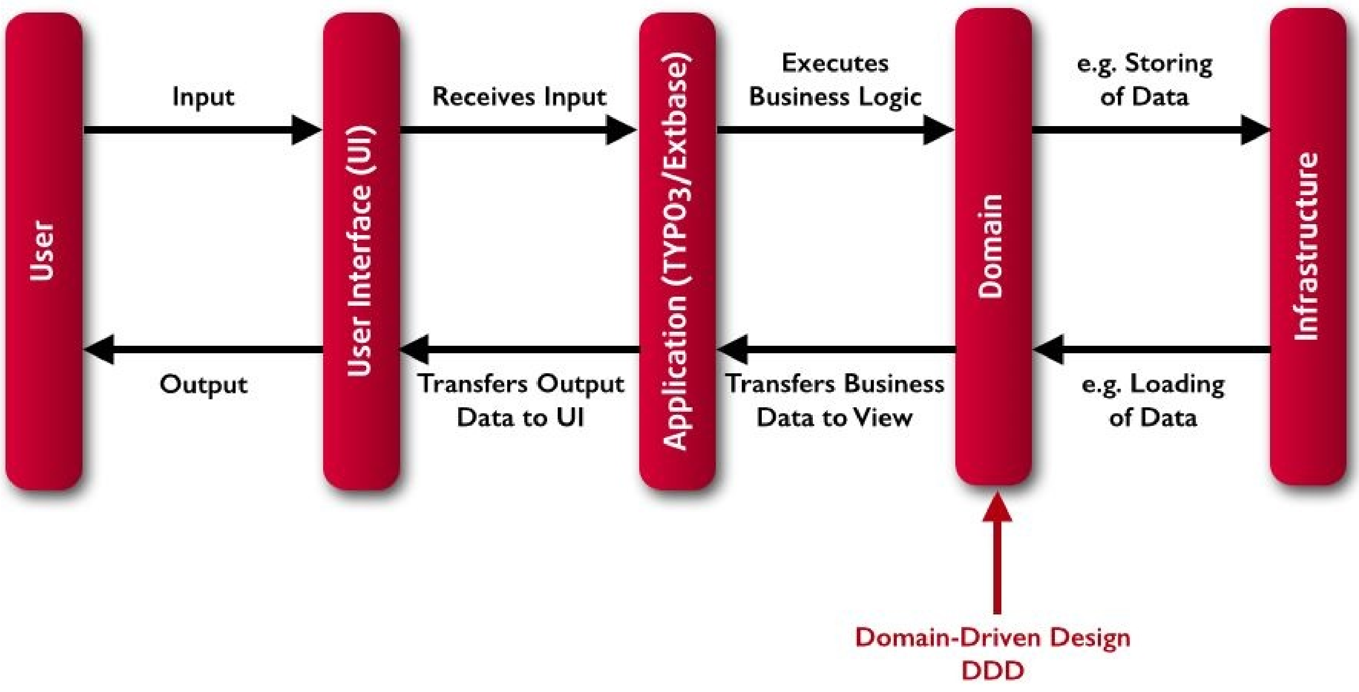


In this context, the “infrastructure” could be templating, input, output, persistence, cookies, AJAX and so on, which has no direct relationship to the problem. We will have to solve these issues too but not as part of DDD. The framework Extbase (which is based on DDD) will take care of many of these tasks for us later. We just have to implement the remaining components.

This is why Extbase does not provide a native solution for an image upload functionality, AJAX handler, etc. These are elements, which have to be implemented in the code using PHP. On the other hand, Extbase enables us to implement the model considerably quicker and in a better quality.

Figure 3.2 visualises, where exactly the DDD sits in a program flow.

Figure 3.2. Domain Driven Design in a vertical architecture



3.2. The Domain Model

A central requirement of DDD dictates that the design of the software is to be made through a model. A model is basically a description of the reality, simplified and focussed on specific purposes. This means that the model represents a plan of objects (and properties) included and their relations to each other. There are no strict rules for a model development but the following guidelines should be taken into account:

- Modelling is the most important process of the entire development and requires a great deal of thought.
- The modelling process should be a collaboration between the developer, the domain expert (the client) and the service provider (e.g. project leader or consultant).
- “Proxy-experts” should not take part in the modelling process but actual domain experts should. A proxy-expert could be a marketing manager who does not know the products in detail for which the software is built but gained their knowledge from a domain expert.
- Modelling is an agile and interactive process. This means that the process does not end at one point necessarily but can be picked up and continued during the project’s lifetime. Due to the fact that the client is an important part of this process, they must contribute the time required.
- The modelling process should be conducted in a way where no technical or intellectual obstacles arise. Therefore no electronic devices and modelling languages should be used at all. A paper and a pencil are the perfect tools.
- The result of the modelling process is a model, that every stakeholder understands straight away.
- The model shows all objects, their properties and methods as well as the relations between these objects.

3.2.1. Ubiquitous Language

A *ubiquitous language* (UL) is a central element to ensure that everyone speaks the same language during the modelling process. Assumed simple or clear terms are quite often misunderstood by team members. Thus it is mandatory that everyone speaks the same language. The focus is on objects, properties, methods and relations used in the domain. Later we will learn that every directory, file and class name follows this principle.

UL leads to the model, which leads to the implementation. In case of a change or adjustment in one of these phases, the other two must be amended accordingly.

As a basic principle, the process must use the language of the domain and as a result of that, creates the basis of the UL. A system for pharmacists should likely use Latin. A software for a Bavarian farmer could even be in Bavarian German, preconditioned, all stakeholders understand the language.

In the past, it was generally agreed that code must be written in English. With DDD, code must be readable by the client as well as the service provider, which means that the language of the domain is the decisive language. In addition, the code is developed to tackle the problem of the client, not the problem of a developer who does not know the domain's language.

Pragmatic Approach

Some developers rightly fear to write their code in Bavarian German. For example, the argument that PHP possibly has issues with Arabic script is also valid. A pragmatic and proven compromise would be that the communication with the domain expert uses the main language but developers use English when communicating with each other.

Figure 3.2 shows a possible, simplified outcome of this step: a glossary.

Table 3.1. Ubiquitous Language Glossary

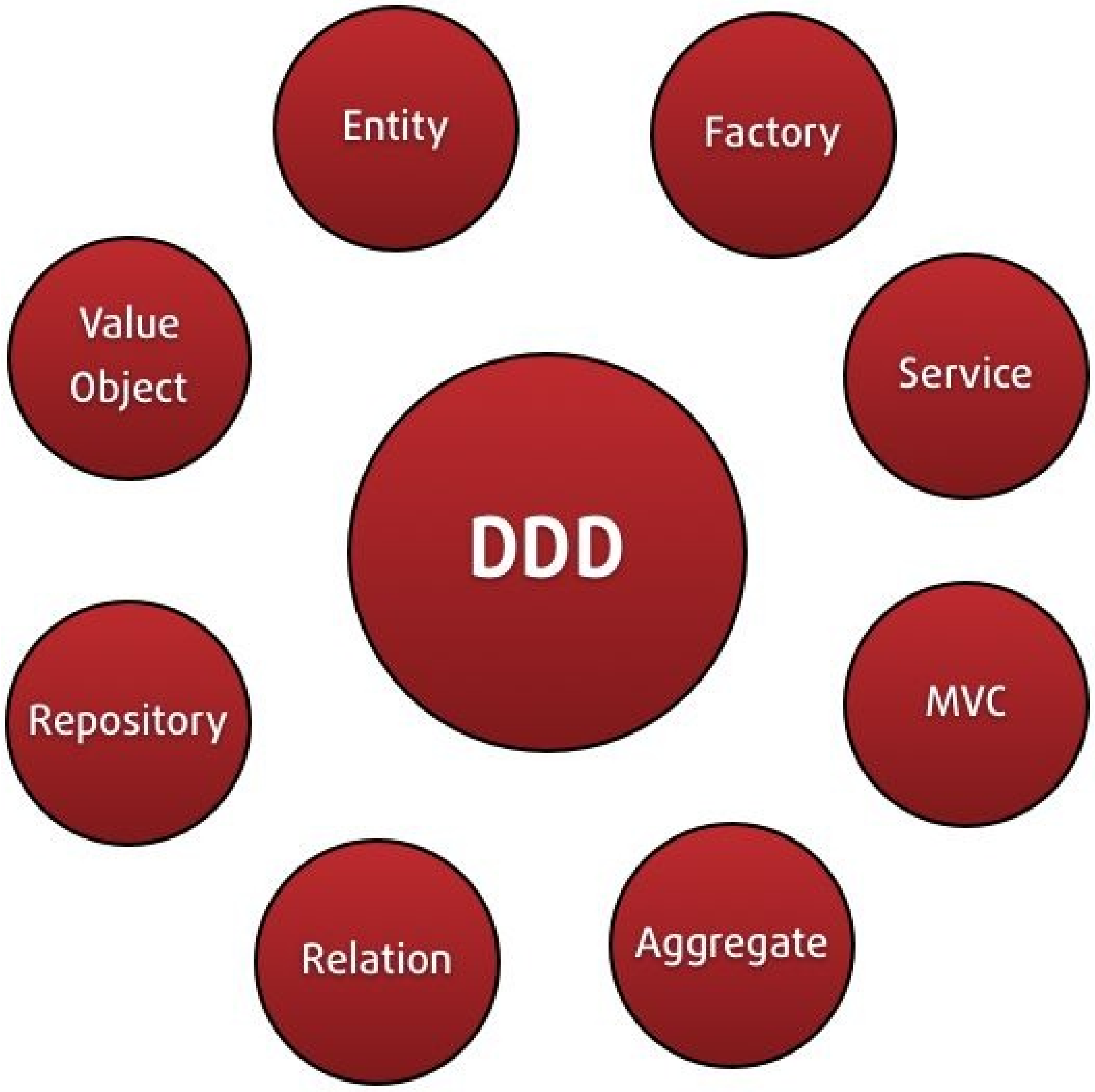
Term	**German**	**Description**	**Important for model**
Order	Bestellung	sum of positions	yes
Row	Posten	unit (consists of product, amount and price)	yes
Warehouse	Lager	location where products are stored (limited capacity)	no
Product	Produkt	physical or virtual article	yes
Invoice	Rechnung	detailed list of ordered products	yes

English, but stakeholder have agreed on using German as the modelling language (e. g. because the organisation is mainly located in Germany).

3.2.2. Building Blocks of DDD

At this point, we have the vocabulary to develop the model but the building blocks are still missing. Not too many of these building blocks exist, relevant for Extbase.

Figure 3.3. Building blocks of DDD



Entity

An *Entity* is a domain object with a global and consistent identity. This could be an unambiguous assignment of a person for example. In order to get the assignment, a unique ID or a combination of multiple fields is required. With the key, the object can be addressed clearly without ambiguity.

This is comparable with databases where a data set is determined by its UID, UUID or a combination of keys. A marriage for example, changes “properties” (e.g. last name or address) but the person remains the same. The passport number or the fingerprint determine the identity.

In order to change an Entity, the system has to find it first, then update it and then store it again.

Value Object

In contrast to an Entity, this domain object does not feature a global identity but is defined by the sum of the properties.

Entities are addressed by `WHERE uid=xy` for example, whereas Value Objects queried by using `WHERE name="red" AND hex="#C00"`. In this case, Extbase reads the UID of the first data set found and internally uses this for further operations.

Changing properties means changing the object so that the object would not be the same any more. A colour could be a *Value Object* for example. If it had the name `red` and the colour code `#c00`, you could not simply change the name to `blue` without meaning a new or different colour. Therefore Value Objects are immutable (not changeable).

When a Value Object is to be changed, it will be discarded and a new one with the new properties is created. In this case, a relation, which possibly exists, is discarded, too.

In general, a Value Object should not have too many properties. From a technical point of view, an index should be stored across columns and this index can hold 1024 bytes maximum, which would already exceed 4 fields of type `varchar(255)`.

Service

While Entities and Value Objects are fundamental components of DDD, objects are required to communicate with each other. This falls in the responsibility of *Services* (e.g. as service classes), which process Entities and Value Objects as their input and/or output. Typical examples are the distance between two address objects or a money transfer between two account objects.

Factory

Factories allow the creation of domain objects in decentralised specific factory objects. This is useful if either the creation is complex (and requires associations for example, which are not required by the domain object) or the specific creation of the domain object needs to be replaced at run time.

Regarding our car object, a Factory could instantiate four wheel objects, link them with a car object and return them later.

Repository

Due to the fact that technical details about the persistence (storage of data for a long period of time) must not occur in the UL and additionally, DDD eludes everything about the infrastructure, databases or any similar storage mechanisms do not exist in DDD.

Therefore, so-called *Repositories* have been introduced. A Repository is a kind of black box that makes objects persistent and allows them to search for objects.

A Repository is always assigned to an object, which means, a car Repository finds cars, a steering wheel Repository finds steering wheels – although, both are stored in the same database from a technical perspective.

Basically, a steering wheel could be found via the car Repository (and vice versa) but only if a relationship between both exists. Furthermore, the object that is assigned to the Repository has to be used.

You could say that your own Repository always allows an independent access to the corresponding object. If a different Repository is used, a relation to the object, which you are looking for, must exist in order to find it.

Aggregate

An *Aggregate* acts as an access control point. Imagine a nightclub, where all the objects inside such as the dance floor, spirituous beverages or spotlights are protected from the outside. In theory, it would be possible to “access” the beverages (object) but the physical wall of the property aggregates the access. Only one spot allows us to penetrate this barrier, the entrance, which is guarded by a bouncer, who can grant access or not. In this case, the bouncer is called *Aggregate Root*.

From a technical point of view, this means for Extbase that all objects, which are also Aggregate Roots, there must be a Repository and a Controller (including appropriate Actions).

Relation

A *Relation* builds a connection between two objects. The following four types of Relations exist in Extbase:

1:1

The 1:1 relation indicates, that for one object, exactly one other object exists. For example, a website owner should only operate one Blog by definition and every Blog should have one operator only.

1:n

The 1:n relation indicates, that for one object, an arbitrary number of other objects exists. A Blog could have a number of entries but an entry belongs to one Blog only.

n:1

The n:1 relation indicates, that an arbitrary number of objects exists for exactly one object. A Blog post for example has one author only but an author could have written an arbitrary number of posts.

m:n

The m:n relation indicates, that an arbitrary number of objects can have a relation to an arbitrary number of other objects. A Blog post could have an arbitrary number of tags but tags can also be assigned to an arbitrary number of posts.

MVC

Domain-Driven Design requires a *Layered Architecture*, which is implemented in Extbase by a MVC framework. MVC stands for:

Model

Business logic and business data. Strictly speaking, this layer should be called *domain*.

View

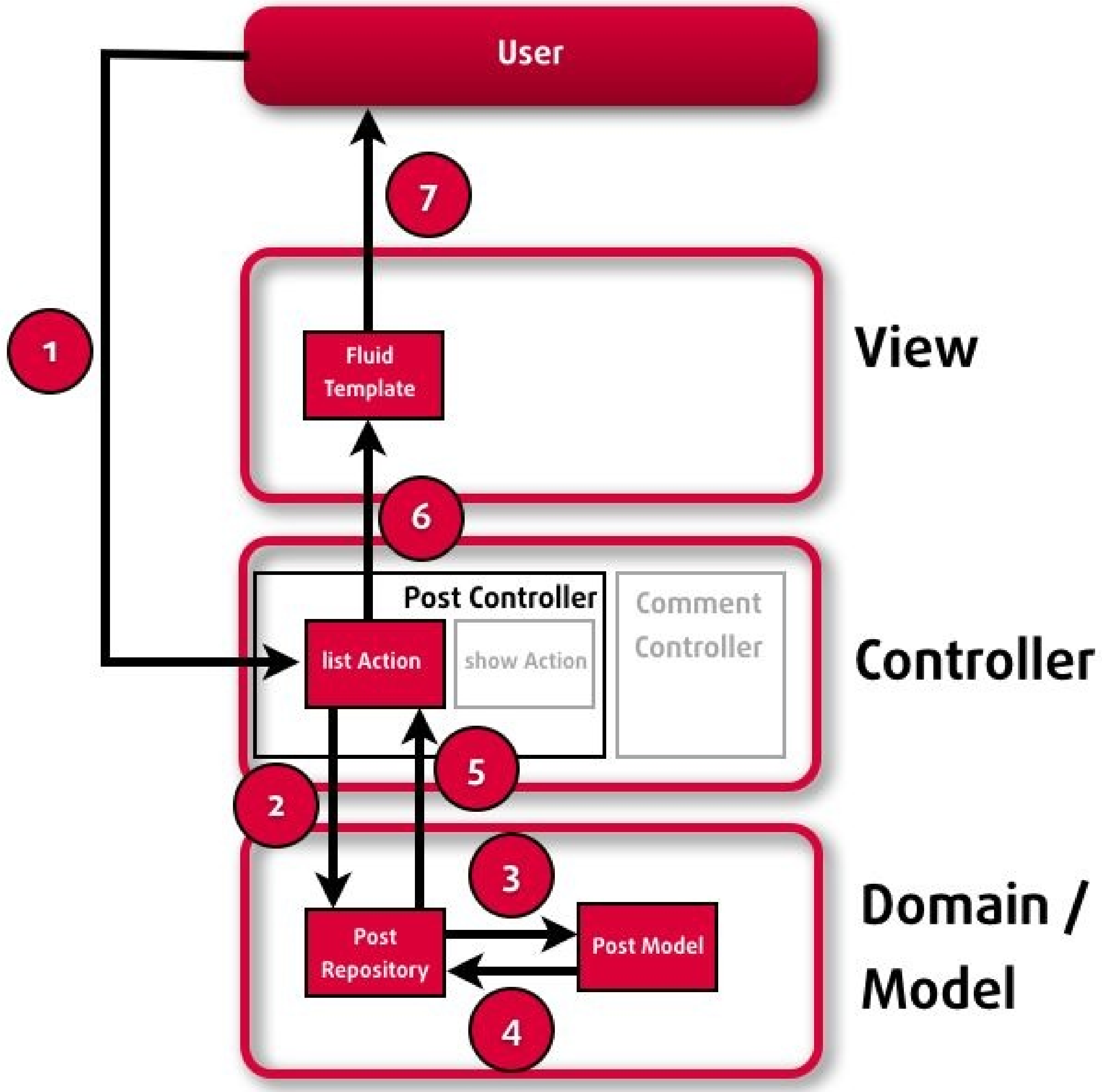
In Extbase, the view (visual appearance or template layer) is implemented by using templates and the underlying templating engine Fluid. Although other views such as Smarty, plain text, XML, JSON, etc. can be used.

Controller

Every request executes the controller first of all. Extbase uses a *slim controller* concept. This means that this component should only contain logic to process the request and control the application – no business logic (model) and no view logic (view) if practical. The controller is diverted into *actions*, which take care of the controlling tasks.

But what does a typical request in Extbase look like?

Figure 3.4. MVC in Extbase



1. First, the user triggers a request, which arrives at the Extbase system. In this case, the user accesses a list of all posts of a Blog. Thus the request hits the *list Action* of the Post Controller.
2. Now the Action queries the domain and accesses the appropriate Post Repository.
3. The Repository passes the data to the domain model and executes any business logics if applicable.
4. The data (technically speaking the QueryResult object only, which is a huge SQL object) is returned to the Repository.
5. The Repository passes the data on to the list Action.
6. …and further to the View, which takes care of the display.
7. Finally, the complete rendered HTML is transferred to the user.

3.3. Structuring DDD

How can the model be kept “clean” respectively and how should it be structured properly? This is particularly important when different developers are working on the domain and/or the domain model become significantly large.

The following concepts exist:

- Bounded Context
- Context Maps
- Core Domain
- Shared Kernel

3.3.1. Bounded Context

The *Bounded Context* (BC) describes the boundaries of the domain in relation to:

- implementation details
- intended purpose
- team classification

A *Context* is initially a specific area of accountability. Assuming this is strictly separated from other areas, you get a *Bounded Context*, a defined environment of a domain.

Let's use an e-commerce shop to illustrate the principle: a product in a shopping cart has a specific context. Certain attributes are important, such as the image, headline, short description. When you access the product in the detail view, further properties appear: a video, 3D images, reviews, etc. As soon as the shipping is calculated, other attributes become relevant, e.g. size, weight, shipping location. The context changes depending on the task.

Almost every non-trivial domain or application uses multiple models. The more important it becomes to draw boundaries where each single model has its own structure and purpose. It should be clear at all times, which problem you want to solve, where the boundaries are and how the components communicate with each other. Eric Evans stated:

> A "Bounded Context" delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other contexts. Within that context, work to keep the model logically unified, but do not worry about applicability outside those bounds. In other contexts, other models apply, with differences in terminology, concepts and rules, and different dialects of the "Ubiquitous Language". By drawing an explicit boundary, you can keep the model pure, and therefore potent, where it is applicable. At the same time you avoid confusion when shifting your attention to other "Contexts". Integration across the boundaries necessarily will involve some translation, which you can analyze explicitly.
>
> — Eric Evans *Domain Driven Design: Tackling Complexity in the Heart of Software*

Every BC includes at least one domain model with a number properties. We will see an example of a Bounded Context in Chapter 17.

3.3.2. Context Map

Context Maps describe the boundaries and interfaces of the Bounded Context as a general plan – from a bird’s eye view to some extent.

This prevents crossing the frontiers of the Bounded Context.

Context Maps clarifies the models that are used at which place in the system. In addition to that, Context Maps provide an overview of all models, their boundaries and interfaces. As a consequence, contexts do not grow into other context areas and the communication between contexts happen through well-defined interfaces.

3.3.3. Core Domain

Every system is divided into three areas:

Core Domain
: This is the most valuable part of the domain model – the part that is of most benefit and business value. Other parts of the domain model aim to support the core domain and expand it with less important functionality. During the modelling, great attention should be paid to the code domain and the best developers should be aware of this.

Generic Subdomains
: These implement generic functionality such as the handling of time zones.

Supporting Domains
: Support domains belong to the circumference of the use case and support the core domain. Without this functionality, the main business is still possible.

For an insurance company, this could be mapped as follows:

Core Domain
: insurance

Supporting Domain
: client’s portfolio

Generic Domain
: time sheet of the insurance agent

3.3.4. Shared Kernel

The *Shared Kernel* is a part of the core domain, which is used by various components across the system. This is useful, if components are only loosely connected with each other and the project is too large to be handled by one team. The shared Kernel is developed by all project teams who intend to use it. This requires appropriate coordination and integration efforts. Therefore a Shared Kernel can be understood as the intersecting set of all Bounded Contexts.

Chapter 4. Overview of Extbase

We will start with a tour through the simple extension `efempty`, in order to gain an overview of the Extbase and Fluid processes happening in the background.

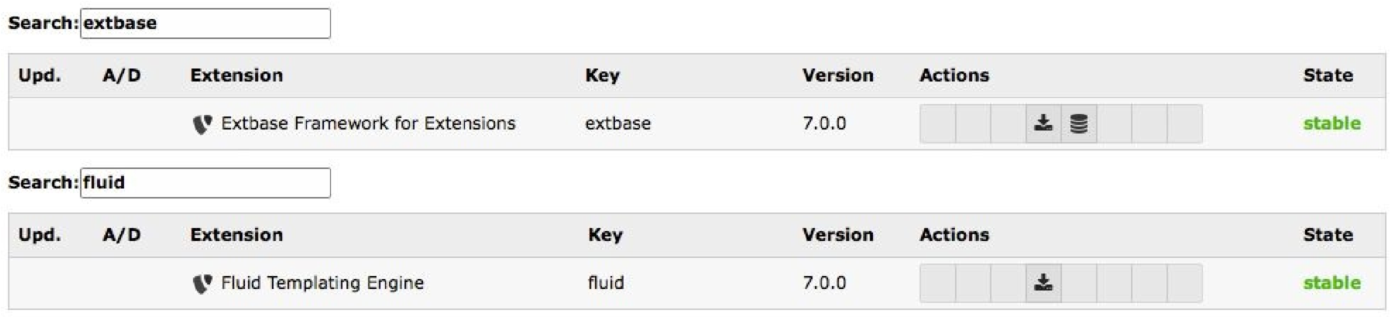
4.1. Installation of Extbase & Fluid

Since TYPO3 CMS version 6.0, the extensions Extbase and Fluid are an inherent part of TYPO3 as system extensions, which can not be uninstalled. This is hardly surprising because more and more system extensions are based on Extbase and Fluid.

However you should always ensure that these packages are installed correctly.

Firstly, go to the *Extension Manager* and check that both extensions are installed in their latest version. If TYPO3 CMS 6.2 LTS is used, Extbase and Fluid show the version number 6.2.0 as well. In TYPO3 7.0.0, Extbase and Fluid show version 7.0.0 accordingly, etc.

Figure 4.1. Extbase and Fluid in TYPO3's Extension Manager



Search: extbase

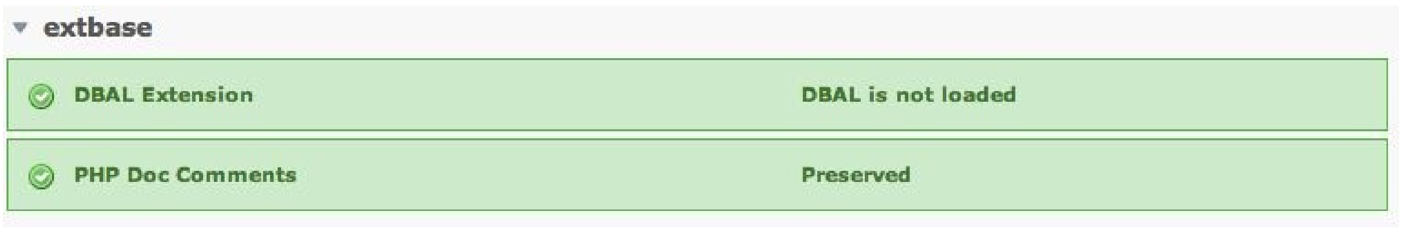
Upd.	A/D	Extension	Key	Version	Actions	State
		Extbase Framework for Extensions	extbase	7.0.0		stable

Search: fluid

Upd.	A/D	Extension	Key	Version	Actions	State
		Fluid Templating Engine	fluid	7.0.0		stable

Secondly, check if annotations remain and have not been removed from the source code (e.g. because they are treated as comments). Go to module SYSTEM → Reports and select Status Report from the dropdown box at the top. There should be two green checks and no warnings/errors in section `extbase` (if you see *DBAL Extension* only and no other checks, that should be fine).

Figure 4.2. Extbase Reports



extbase

DBAL Extension	DBAL is not loaded
PHP Doc Comments	Preserved

4.1.1. Preserve PHP Doc Comments

If TYPO3 reports an issue with `PHP Doc Comments` (red error message), you will have to fix this problem before continuing.

For a production system it is often recommended to use a PHP Opcode Cache such as eAccelerator.[12] These add-ons cache PHP scripts in their compiled state and significantly increase their performance. By default, eAccelerator does not store comments in PHP scripts to the cached files.

As explained before, Extbase requires these comments for important control information and therefore comments must be preserved by eAccelerator. This can be done by using the configuration option `--witheaccelerator-doc-comment-inclusion`.

Below is the complete installation of eAccelerator:

- Download the source code of eAccelerator, extract the archive and change to the directory.
- Execute the command `phpize` to tune eAccelerator's source code to your specific PHP version.
- Prepare compiling the code and tell eAccelerator to preserve comments by using the following command and paremeter: `./configure --witheaccelerator-doc-comment-inclusion`.
- Now you can compile eAccelerator by executing the command `make`
- Finally, install eAccelerator by executing the command `make install`, which requires `root` privileges.

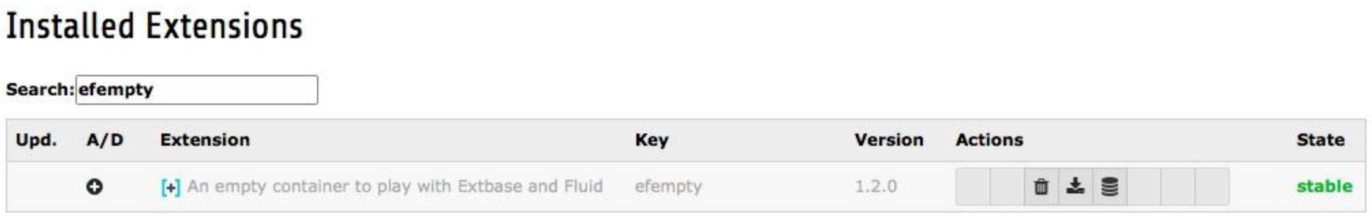
You may need to adjust the configuration of PHP to load eAccelerator.

4.2. Installation of Extension efempty

The extension `efempty` has been developed for didactic purposes and contains a minimal extension which enables developers to become acquainted with Extbase and Fluid. The extension can also be used as a foundation for your own, more complex developments by simply adjusting the PHP files.

Go to the Extension Manager and install the extension `efempty`. Make sure you are using the latest version from the TER list and at least version 1.1.1. Earlier versions are not optimised for TYPO3 CMS 6.0 or higher and could cause problems.

Figure 4.3. Extension efempty in TYPO3's Extension Manager



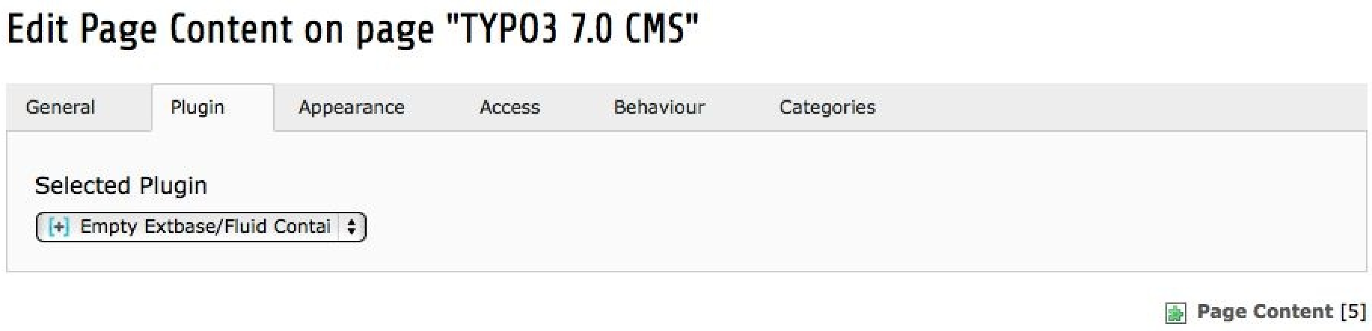
Installed Extensions

Search: efempty

Upd.	A/D	Extension	Key	Version	Actions	State
	⊕	[+] An empty container to play with Extbase and Fluid	efempty	1.2.0		stable

In the next step, we will place the extension on a page. Go to *WEB → Page* and add a content element of type "General Plugin" (tab `Plugins`) to columns *Normal*. Select the plugin `Empty Extbase/Fluid Container` from the dropdown box in tab *Plug-In*.

Figure 4.4. Select the plugin



Edit Page Content on page "TYPO3 7.0 CMS"

Selected Plugin: [+] Empty Extbase/Fluid Contai

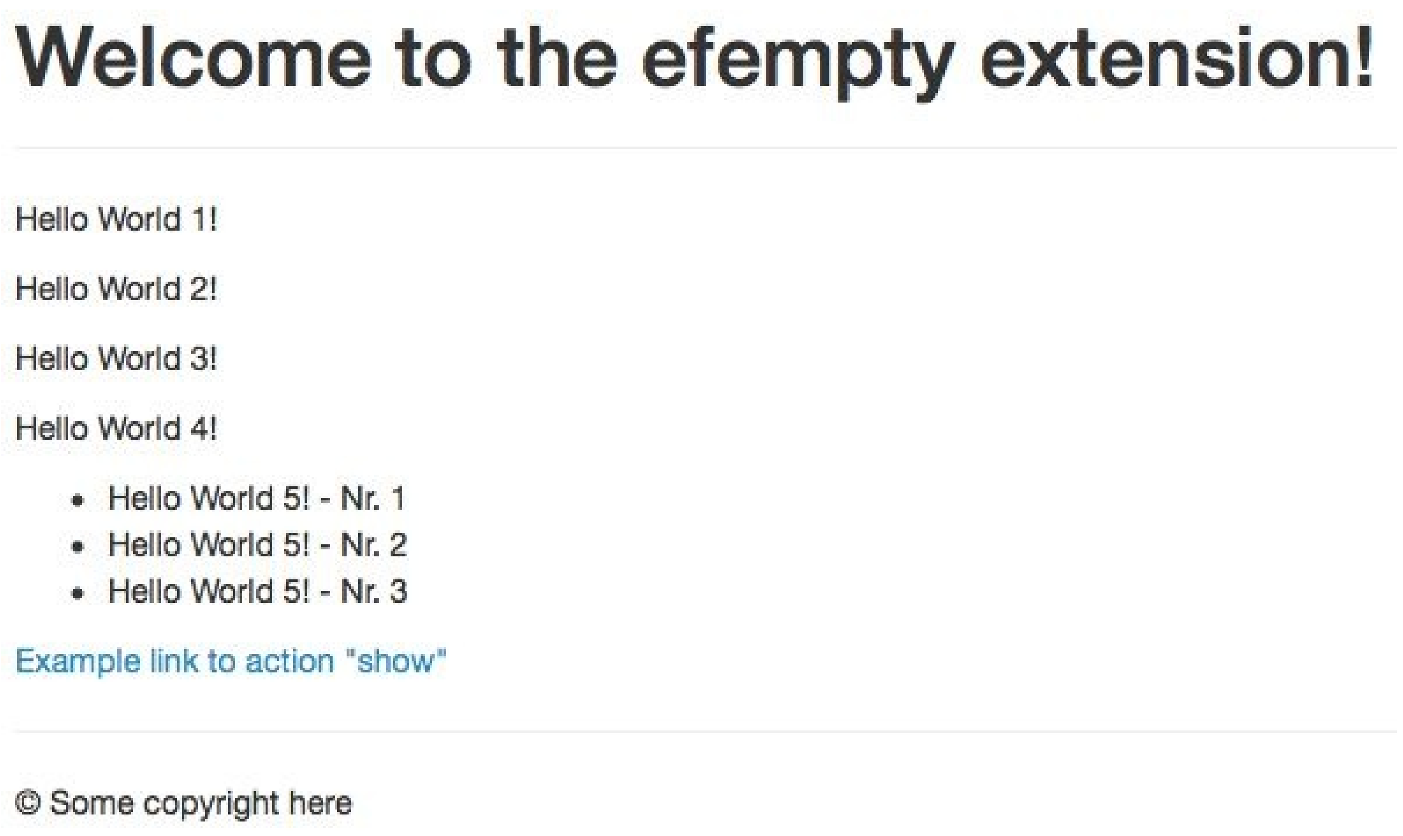
Assuming that you are using a pre-configured system, you can already see the content in the frontend. Otherwise you will have to create a TypoScript template. Go to module *Template*, select *Info/Modify* from the dropdown box at the top and click the button to create a template for a new site. Following the link to *edit the whole template record* and enter the following code into the Setup field:

```
page = PAGE
page.10 < styles.content.get
```

Now switch to tab *Includes* and choose *CSS Styled Content (css_styled_content)* from *Available Items* (right) under the *Include static (from extensions)* section. The item appears in the left field.

Finally, save the template record by using the *Save and close document* icon at the top (disk with X). When accessing the frontend, you should see a page similar to Figure 4.5.

Figure 4.5. Output of efempty



Welcome to the efempty extension!

Hello World 1!

Hello World 2!

Hello World 3!

Hello World 4!

- Hello World 5! - Nr. 1
- Hello World 5! - Nr. 2
- Hello World 5! - Nr. 3

Example link to action "show"

© Some copyright here

4.3. Tour Through Extension efempty

The following sections describe in detail what happens in the background when Extbase, Fluid and also TYPO3 CMS generates this output.

4.3.1. Files ext_emconf.php and ext_icon.gif

The file `ext_emconf.php` contains the configuration of the extension. You will find a complete reference of all configuration options in the appendix.

```
<?php

/***************************************************************
 * Extension Manager/Repository config file for ext "efempty".
 *
 * Auto generated 18-01-2015 09:01
 *
 * Manual updates:
 * Only the data in the array - everything else is removed by next
 * writing. "version" and "dependencies" must not be touched!
 ***************************************************************/

$EM_CONF[$_EXTKEY] = array (
        'title' => 'An empty container to play with Extbase and Fluid',
        'description' => 'This extension just contains a Controller (Start)
        'category' => 'plugin',
        'shy' => 0,
        'version' => '1.2.0',
        'dependencies' => '',
        'conflicts' => '',
        'priority' => '',
        'loadOrder' => '',
        'module' => '',
        'state' => 'stable',
        'uploadfolder' => 0,
        'createDirs' => '',
        'modify_tables' => '',
        'clearcacheonload' => 0,
        'lockType' => '',
        'author' => 'Patrick Lobacher',
        'author_email' => 'patrick@lobacher.de',
        'author_company' => 'LOBACHER.',
        'CGLcompliance' => NULL,
        'CGLcompliance_note' => NULL,
        'constraints' =>
        array (
                'depends' =>
                array (
                        'php' => '5.3.7-0.0.0',
                        'typo3' => '6.0.0-7.9.99',
                ),
                'conflicts' =>
                array (
                ),
                'suggests' =>
                array (
                ),
        ),
);
```

```
?>
```

Some of the information from this file (e.g. `title`, `description`, `version`, etc.) is visible in the TER (when you are searching for this extension) as well as in TYPO3’s Extension Manager. This also includes the logo (file: `ext_icon.gif`).

When clicking on *Install*, a directory `efempty` is created under `typo3conf/ext/` and the extension files are copied into this. Afterwards the extension key efempty is written to the file `typo3conf/PackageState.php` (key: `packages`).

```
<?php

return array (
  'packages' =>
  array (
    ...
    'efempty' =>
    array (
      'state' => 'active',
      'packagePath' => 'typo3conf/ext/efempty/',
      'classesPath' => 'Classes/',
    ),
    ...
  ),
  'version' => 4,
)
 ?>
```

At this point, the extension has been successfully installed and is ready to use as a content element on a page.

4.3.2. File ext_tables.php

In order to allow the extension to be added to a page, file `ext_tables.php` is analysed:

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

\TYPO3\CMS\Extbase\Utility\ExtensionUtility::registerPlugin(
    'Lobacher.' . $_EXTKEY,
    'Showcase',
    'Empty Extbase/Fluid Container'
);

?>
```

Initially, a check is conducted to see if the file is executed directly. In this case, the script is terminated immediately.

Then, the first Extbase-API-function is called. Class `\TYPO3\CMS\Extbase\Utility\ExtensionUtility` also describes the location in the file system:

- The first two elements represent the vendor name. The TYPO3 project is allowed to use two elements (`\TYPO3\CMS`) – extension developers may only use one element.
- The third element represents the extension name – in this instance: `Extbase`. TYPO3 searches for the extension key in directory `typo3conf/ext/` first, then in `typo3/ext/` and finally in `typo3/sysext/`, where it discovers the system extension Extbase.
- At that point a folder structure under `extbase` exists. Due to the fact that in Extbase all classes are stored inside the directory `Classes`, this can be left out. The directory Utility is consequentially accessible as `typo3/sysext/extbase/Classes/Utility`.
- If further paths would be stated, they would appear as additional elements. An exception is the last element: `ExtensionUtility`. This element defines the class file with `.php` appended (`ExtensionUtility.php`).
- This file contains a PHP class called `ExtensionUtility`.

The static function `registerPlugin` is called in this class with the following parameters:

`'Pluswerk.' .$_EXTKEY`
: The vendor name gets appended by the extension key `efempty`, which in theory would manage multiple extension keys in one instance (however this is not possible at this point in time due to the restrictions of TYPO3 CMS).

`Showcase`
: Euphonic names such as `pi1`, `pi2` or `pi3` have been used in pre-Extbase extensions. Now, arbitrary names can be used. This name acts as a plugin key, which we will use later on.

`Empty Extbase/Fluid Container`
: This description is shown in the list of available plugins when an extension is added

as a content element. It is also possible to link to a language file (XLIFF) in order to achieve multi-language support.

The appropriate section in file `typo3\sysext\extbase\Classes\Utility\ExtensionUtility.php` as follows:

```
<?php
namespace TYPO3\CMS\Extbase\Utility;

class ExtensionUtility {

   /**
    * Register an Extbase PlugIn into backend's list of plugins
    * FOR USE IN ext_tables.php FILES
    *
    * @param string $extensionName The extension name (in UpperCamelCase) or
    * @param string $pluginName must be a unique id for your plugin in Upper
    * @param string $pluginTitle is a speaking title of the plugin that will
    * @param string $pluginIconPathAndFilename is a path to an icon file (re
    * @throws \InvalidArgumentException
    * @return void
    */
   static public function registerPlugin(
         $extensionName,
         $pluginName,
         $pluginTitle,
         $pluginIconPathAndFilename = NULL
      ) {
      ...
```

This method allows for a fourth parameter. This is the icon of the plugin. Without this parameter, the plugin uses the icon on the extension `ext_icon.gif`.

Multiple plugins in one extension

A plugin is ultimately a unit of functions that belong together. In an online shop, this could be a shopping cart, a product view and a product list. This can be achieved by developing three plugins as one extension. Simply add multiple register Plugin statements one after another but ensure that the plugin key is unique across them.

4.3.3. ext_localconf.php

While the file `ext_tables.php` is responsible for integrating the plugin as a content element in the backend, the file `ext_localconf.php` ensures that the plugin can be accessed in the frontend.

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

\TYPO3\CMS\Extbase\Utility\ExtensionUtility::configurePlugin(
    'Pluswerk.' . $_EXTKEY,
    'Showcase',
        array(
                'Start' => 'index,show',
        ),
        array(
                'Start' => 'index,show',
        )
);

?>
```

As before, a static function in class `ExtensionUtility` is called – this time the method `configurePlugin`, which *configures* the plugin:

`'Pluswerk.' . $_EXTKEY`
: The vendor name gets appended by the extension key as before.

`'Showcase'`
: This is the plugin name, which must match the name stated in file `ext_tables.php`.

First Array
: This array stores line-by-line the controllers (here: `Start`) as associative keys and available actions as comma-separated values (here: `index` and `show`). Only these combinations are valid. In the case that an action is called, which is not listed in this array, it falls back to the default controller action. In our example the first controller (`Start`) and the first action in the list (`index`). The same happens if no controller-action-combination has been used in the request (this was the case when we accessed the frontend above).

Second Array
: Another array with the same structure can be stated as a second array. The combinations included here are not cached when the content gets rendered. Therefore, this array is always a subset of the first one. During development, it is recommended to keep both arrays the same in order to prevent caching problems.

The appropriate section in file `typo3\sysext\extbase\Classes\Utility\ExtensionUtility.php` as follows:

```
<?php
```

```
namespace TYPO3\CMS\Extbase\Utility;

class ExtensionUtility {
   /**
     * Add auto-generated TypoScript to configure the Extbase Dispatcher.
         *
         * When adding a frontend plugin you will have to add both an entry
         * of tt_content table AND to the TypoScript template which must ini
         * Since the static template with uid 43 is the "content.default" an
         * used for rendering the content elements it's very useful to have
         * adding the necessary TypoScript for calling the appropriate contr
         * It will also work for the extension "css_styled_content"
         * FOR USE IN ext_localconf.php FILES
         * Usage: 2
         *
         * @param string $extensionName The extension name (in UpperCamelCas
         * @param string $pluginName must be a unique id for your plugin in
         * @param array $controllerActions is an array of allowed combinatio
         * @param array $nonCacheableControllerActions is an optional array
         * @param string $pluginType either \TYPO3\CMS\Extbase\Utility\Exten
         * @throws \InvalidArgumentException
         * @return void
         */
        static public function configurePlugin(
              $extensionName,
              $pluginName,
              array $controllerActions,
              array $nonCacheableControllerActions = array(),
              $pluginType = self::PLUGIN_TYPE_PLUGIN
           ) {
           ...
```

The fifth parameter allows us to define if the extension should be included as a plugin (`PLUGIN_TYPE_PLUGIN` – this is the default) or as a content element (`PLUGIN_TYPE_CONTENT_ELEMENT`).

Interna

Method `configurPlugin()` initiates the following actions.

Firstly, a configuration is added to the array `$GLOBALS['TYPO3_CONF_VARS']`. For every controller, an entry is being made (one for “normal” actions and one for non-cachable actions):

```
$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['extbase']['extensions'] ['Efempty'][

$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['extbase']['extensions'] ['Efempty'][
```

Afterwards some default TypoScript is written to the standard configuration:

```
plugin.tx_efempty {
        settings {
        }
        persistence {
                storagePid =
                classes {
                }
        }
        view {
                templateRootPaths {
                        #example: fooKey = EXT:bar/foo
                }
                layoutRootPaths {
                        #example: fooKey = EXT:bar/foo
                }
                partialRootPaths {
                        #example: fooKey = EXT:bar/foo
                }
                 # with defaultPid you can specify the default page uid of t
                defaultPid =
        }
}
```

In TYPO3 CMS versions prior 6.2, the keys were named `templateRootPath`, `layoutRootPath`, `partialRootPath` (without the trailing *s*). They still work but are classified as “deprecated”.

In fact, all the values are empty because Extbase assumes that the keys exist. Custom TypoScript fills these values in later. At last, the rendering definition is created (for the type `PLUGIN_TYPE_PLUGIN`) and written into the TypoScript setup.

```
tt_content.list.20.efempty_showcase = USER
tt_content.list.20.efempty_showcase {
   userFunc = TYPO3\CMS\Extbase\Core\Bootstrap->run
   extensionName = Efempty
   pluginName = Showcase
   vendorName = Pluswerk
}
```

Here we can see clearly, how the execute by Extbase happens:

- first, a USER object is created
- the method `run()` of class `\TYPO3\CMS\Extbase\Core\Bootstrap` is called
- three parameters are passed to this method: the name of the extension, the name of the plugin and the name of the vendor

At this point, Extbase is responsible for the execution of the process rather than the TYPO3 CMS framework.

Assuming the type would be `PLUGIN_TYPE_CONTENT_ELEMENT` instead, the TypoScript would look like:

```
tt_content.efempty_showcase = COA
tt_content.efempty_showcase {
        10 = < lib.stdheader
        20 = USER
        20 {
                userFunc = TYPO3\CMS\Extbase\Core\Bootstrap->run
                extensionName = Efempty
                pluginName = Showcase
                vendorName = Pluswerk
        }
}
```

4.3.4. Controller: Pluswerk\Efempty\Controller\StartController

As described earlier, the first step is to determine the controller. We have not passed any parameters in the request (e.g. `$_GET` or `$_POST`), which lets Extbase use the configuration from file `ext_localconf.php`.

The default controller is defined as Start and Extbase searches for the file `StartController.php` in directory `typo3conf/ext/efempty/Classes/Controller` and includes this automatically.

```
<?php
namespace Pluswerk\Efempty\Controller;

class StartController extends \TYPO3\CMS\Extbase\Mvc\Controller\ActionContro

        /**
         * Initializes the current action
         *
         * @return void
         */
        public function initializeAction() {

        }

        /**
         * Index action for this controller.
         *
         * @return string The rendered view
         */
        public function indexAction() {

                // plain assign
                $this->view->assign('helloworld1', 'Hello World 1!');

                // normal array assign
                $array = array('Hello','World','2!');
                $this->view->assign('helloworld2', $array);

                // associative array assign
                $array = array('first' => 'Hello', 'middle' => 'World', 'las
                $this->view->assign('helloworld3', $array);

                // object assign
                $start = new \Pluswerk\Efempty\Domain\Model\Start();
                $start->setTitle("Hello World 4!");
                $this->view->assign('helloworld4', $start);

                // more object assign
        $obj = array();
        for ($i=1; $i<=3; $i++) {
            $start = $this->objectManager->get('\\Pluswerk\\Efempty\\Domain\
            $start->setTitle('Hello World 5! - Nr. '.$i);
            $obj[] = $start;
        }
                $this->view->assign('helloworld5', $obj);

        }

    /**
     * Index action for this controller.
     *
     * @return string The rendered view
     */
    public function showAction() {

    }
}

?>
```

The class extends from class `\TYPO3\CMS\Extbase\Mvc\Controller\ActionController`, which executes a number of actions in the background and provides many functions.

Three methods (all with the suffix `Action`) build the Action methods of the controller.

The first method `initializeAction` is always called before any other action. This method is a perfect fit for configuration tasks or generic logging for example. In addition, a method called `initializeIndexAction` exists, which is called directly after `initializeAction` and before `indexAction`. We do not need any initialisation here so the method body remains empty.

The Bootstrap successfully identified the controller and searches for the action now. As before: without any `$_GET` or `$_POST` parameters in the request Extbase uses the configuration from file `ext_localconf.php`. The default action is stated in the first row of the array, the first keyword of the string: `index`. This means, Extbase calls method `indexAction`.

View Allocation

The abstract ActionController, which our StartController has been derived from, provides the View, which is declared by the proprty `$this->view`. Just a few methods can be applied to this and `assign('key', 'value')` is one of them, where `value` is assigned to `key`.

Investigating the five directives concludes that:

1. The string `Hello World 1` is assigned to the key `helloworld1`.
2. A numeric array with three elements `Hello`, `World` and `2!` is assigned to the key `helloworld2`.
3. An associative array with three key-value-pairs `first->Hello`, `middle->World` and

`last->3!` is assigned to the key `helloworld3`.

4. An instance of the domain object `\Pluswerk\Efempty\Domain\Model\Start` is assigned to the key `helloworld4`, where the attribute title is set to `Hello World 4!`.
5. Three instances of the domain object `\Pluswerk\Efempty\Domain\Model\Start` are assigned to the key `helloworld5` as arrays.

We will return to this later. For now, we will focus on the domain object first.

4.3.5. Domain: Pluswerk\Efempty\Domain\Model\Start

At the fourth and fifth example of the controller, an instance of the domain object `Start` has been instantiated. In order to understand this more clearly, let us have a look at the domain object class file:

```
<?php
namespace Pluswerk\Efempty\Domain\Model;

class Start extends \TYPO3\CMS\Extbase\DomainObject\AbstractEntity {

        /**
         * Some title.
         *
         * @var string
         */
        protected $title = '';

        /**
         * An empty constructor - fill it as you like
         *
         */
        public function __construct() {

        }


        /**
         * Sets the title
         *
         * @param string $title
         * @return void
         */
        public function setTitle($title) {
                $this->title = $title;
        }

        /**
         * Gets the title
         *
         * @return string The title of the album
         */
        public function getTitle() {
                return $this->title;
        }

}

?>
```

We realise that this domain object is an Entity because it is derived from `\TYPO3\CMS\Extbase\DomainObject\AbstractEntity`. The only property `title` is defined near the top and a “Setter” and “Getter” has been implemented.

4.3.6. Output Via View

Let us go back to the controller. The method ends after the fifth directive. However some content is shown. This is produced by an invisible code at the end of the Action:

```
class StartController extends \TYPO3\CMS\Extbase\Mvc\Controller\ActionContro

        public function indexAction() {
                ...
                $this->view->assign('helloworld5', $obj);

                // Der Code ist normalerweise nicht sichtbar
                return $this->view->render();
        }
}
```

As long as an Action does not have a return value, the method `$this->view->render()` is executed implicitly so that the current Fluid View (respectively the template) is searched and rendered. The output is returned as HTML and TYPO3 displays it.

In Extbase, the template file has its own, specific location: a folder with the same name as the controller is searched in the directory `Resources/Private/Templates`. Inside this folder, a file is read, which it’s name describes the Action in UpperCamelCase and ends with the suffix `.html`, which in our case results in: `typo3conf/ext/efempty/Resources/Private/Templates/Start/Index.html`.

Notation of template files

Names of template files always have to be in UpperCamelCase, even if the Action method is spelled lowerCamelCase. Up to TYPO3 CMS version 4.5, templates could also be named in lowerCamelCase, which is not possible anymore.

Chapter 8 contains an overview of templates, partials and layouts.

```
<f:layout name="defaultLayout" />

<f:section name="content">

<p>{helloworld1}</p>

<p>{helloworld2.0} {helloworld2.1} {helloworld2.2}</p>

<p>{helloworld3.first} {helloworld3.middle} {helloworld3.last}</p>

<p>{helloworld4.title}</p>

<ul>
<f:for each="{helloworld5}" as="helloworld">
        <li>{helloworld.title}</li>
```

```
</f:for>
</ul>

<f:link.action action="show">Example link to action "show"</f:link.action>

</f:section>
```

All tags starting with `<f:` are so called Fluid-ViewHelper. These support the View with functionality and logic.

`<f:layout name="defaultLayout" />` loads a layout file with the name `DefaultLayout.html`, which is expected in the directory `Resources/Private/Layouts/`.

```
<h1>Welcome to the efempty extension!</h1>

<hr />

<f:render section="content" />

<hr />

<p>&copy; Some copyright here</p>
```

A number of outputs happen here and wrapped in the middle a ViewHelper `<f:render section="content" />` renders a section named `content` in the template. A corresponding section exists in the template file, which can be found between `<f:section name="content">` and `</f:section>`.

After that, four paragraphs `<p>...</p>` are rendered, which represent the directives in the controller:

1. A simple string can be output by using `{helloworld1}`. Curly brackets tell Fluid to display the value previously assigned to the key.
2. In order to access values of a numeric array, the key plus a dot plus the appropriate index can be used: `{helloworld2.0}` shows the first element of the array.
3. Similar works for associative arrays, where the array key is used instead of the index: `{helloworld3.first}` shows the element with the key `first` of the array.
4. For objects, the object property is used after the dot. In this case, Fluid not only returns the property but calls the method `getTitle()` of the domain model in the background.

In the fifth example an array of objects has been assigned. These are read by the `<f:for>` ViewHelper, which iterates through the array (`{helloworld5}`) and describes each object as helloworld. Inside the loop, the object can be accessed via `{helloworld}`.

4.3.7. The Show-Action Call

At the end of the template file an additional Fluid-ViewHelper-Link is placed, which refers to the ShowAction:

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

Shown at the frontend as follows:

<a href="index.php?id=1&
    tx_efempty_showcase[action]=show&
    tx_efempty_showcase[controller]=Start&
    cHash=61470f084417e294090bec4f74c42e7d">Example link to action "show"</a
```

Here we see that two `$_GET` parameters exist, which pass control instructions to Extbase. The request also contains a name space `tx_efempty_showcase`, which tells Extbase that the parameters are meant for the extension `efempty` and the plugin Showcase. Additionally, the controller is explicitly set to `Start` and the Action to `show`.

Therefore Extbase calls the `showAction()` of controller `StartController`. The Action itself is empty, which means the template `Resources/Private/Templates/Start/Show.html` is loaded:

```
<f:layout name="defaultLayout" />

<f:section name="content">

This is the content of the show action template.

<hr />

<f:translate key="this.is.a.key">Translated content</f:translate>

<hr />

<f:link.action action="index">Back</f:link.action>

</f:section>
```

Besides that “back” link at the end of the file, the only new element is the `<f:translate>` ViewHelper, whose argument `key` will be searched for in the file `Resources/Private/Language/locallang.xlf` and displayed. The purpose of this is multi-language support. Assuming TYPO3 CMS has been configured to use German (by stating `config.language = de` in the TypoScript setup), the appropriate value of key in file `Resources/Private/Language/de.locallang.xlf` is shown.

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<xliff version="1.0">
        <file source-language="en" datatype="plaintext" original="messages"
                <header/>
                <body>
                        <trans-unit id="this.is.a.key">
                                <source>Example Output from language file</s
                        </trans-unit>
                </body>
        </file>
</xliff>
```

At this point we have reached the end of the first stage of our Extbase tour. The next chapters cover all areas in detail by using a continuous example.

---

[12] http://eaccelerator.net

Chapter 5. Domain Model Creation (Modelling)

As described in Chapter 3 in more detail, the first step for building an extension is the modelling of the domain model.

There are two parts involved in this process, one is the consultation with the client in order to achieve a vocabulary and a simple model (possibly handwritten only). The other is the modelling in Extbase.

5.1. Domain Model Used in this Book

For the remainder of this book we will use the Blog-example, which is well-suited to didactic purposes and this has been proven in hundreds of training sessions. The domain aspect of a Blog is comprehensible and the developer can focus on the code rather than learning and understanding the domain.

5.2. Basic Concept

The vision of our extension should be, that we re-implement the functionality of the `blogger.com` site. The following functions are desirable:

- Creation of an unlimited number of Blogs.
- Then you can create as many posts as you want.
- An author and various tags can be assigned to one post.
- An arbitrary number of comments can be submitted for each post.
- Later, an RSS feedback of recent posts should exist.
- Comments can also be deleted in the backend.
- An additional plugin should always list the last five posts.
- Through a tag-cloud all posts should be displayed which contain this tag.

5.2.1. The Glossary

First, we create the glossary to ensure that every stakeholder of the project has the same understanding of the terminology used. We will use the English language for domain objects because objects in a Blog domain tend to use English too (post, comment, tag).

Blog

> Weblog
>
> Relevant for model: yes

Post

> Article written by an author for a specific Blog at a specific date/time.
>
> Relevant for model: yes

Comment

> Short reply of a post.
>
> Relevant for model: yes

Tag

> Descriptive word (e.g. topic). A post may have an unlimited number of tags.
>
> Relevant for model: yes

Author

> TYPO3 frontend user.
>
> Relevant for model: yes

5.2.2. Creating the Domain Model

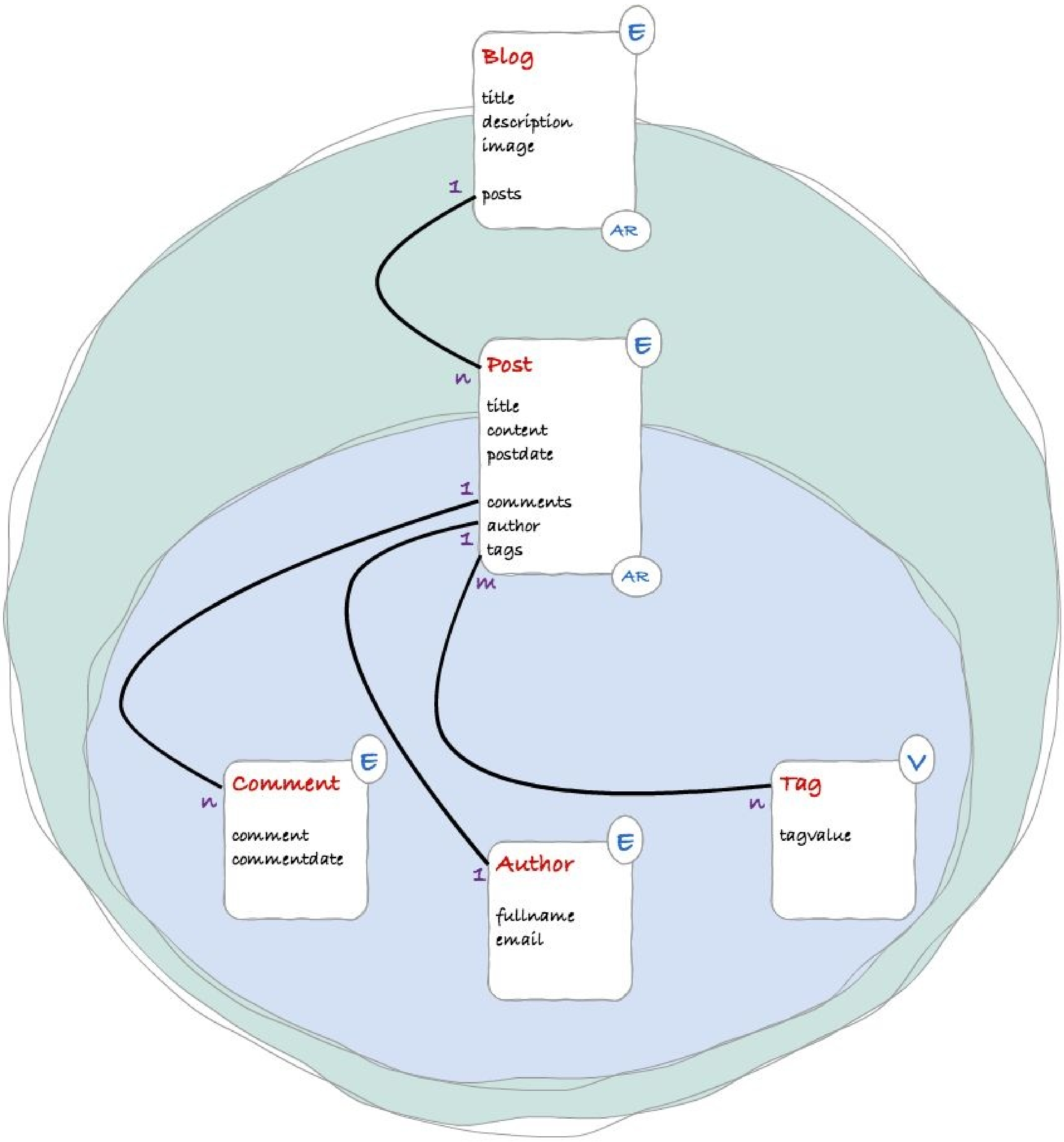
Now we can create the domain model, this could be handwritten but for our purposes we will use a software[13] for better illustration. Due to possible technical barriers (a software tool possibly adds another level of complexity), we recommend to not use electronic tools when drawing the domain model together with the client.

Figure 5.1 shows the following:

- Five domain objects exist (`Blog`, `Post`, `Comment`, `Author` and `Tag`).
- Domain object `Blog` has the properties `title`, `description` and `image`.
- Domain object `Post` has the properties `title`, `content` and `postdate`.
- Domain object `Comment` has the properties `comment` and `commentdate`.
- Domain object `Author` has the properties `fullname` and `email`.
- Domain object `Tag` has the property `tagvalue`.
- Domain objects `Blog`, `Post`, `Comment` and `Author` are of type `Entity` (recognisable by the letter `E` at the top, right-hand-side).
- Domain object `Tag` is of type `Value Object` (recognisable by the letter `V` at the top, right-hand-side).

There are two `Aggregates` – one in the light green area in `Blog` as `Aggregate Root` (recognisable by the letters `AR` at the bottom, right-hand-side) and another one in the light blue area in `Post` as `Aggregate Root`.

Figure 5.1. Domain Model of the Blog Example



- There is also a *1:n relation* between `Blog` and `Post` named `posts`, which means a Blog can contain an arbitrary number of posts but a post can only belong to one Blog.
- Another *1:n relation* exists between `Post` and `Comment` named `comments`, which means a post can contain an arbitrary number of comments but a comment only belongs to one post.
- A *1:1 relation* between `Post` and `Author` named `author` indicates, that a post can be published by only one author and an author can only publish one post (we will change this to a *n:1 relation* as we go along).
- A *n:m relation* between `Post` and `Tag` named `tags` means that a post can contain an arbitrary number of tags and a tag can be assigned to an arbitrary number of posts.

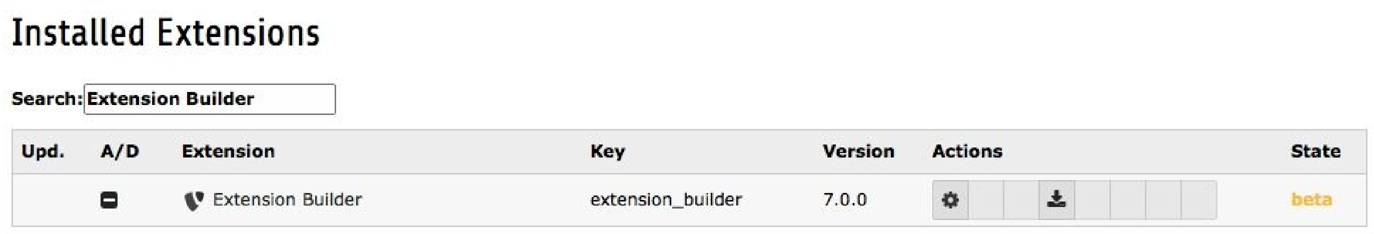
5.3. Modelling in Extbase - The Extension Builder

Our aim is to have the model in Extbase of course. A TYPO3 extension named *Extension Builder* supports us in this transformation process so install this extension via the Extension Manager in TYPO3 and reload the backend as a precaution.

Do not over-rely on the Extension Builder!

The Extension Builder is (currently) a project maintained by just one or two developers (who do a fantastic job by the way), which is convenient and using it saves a lot of time. However sometimes the code generated includes some minor issues because there is a lack of quality assurance and management due to insufficient man power. These issues can be located and fixed easily, if you know what code is being generated and exactly what the code should do. Therefore, you should not overly-rely on the Extension Builder but try to understand the basic concept. In general, you should be able to write your code even without the Extension Builder or at least understand its entirety.

Figure 5.2. Installation of the Extension Builder



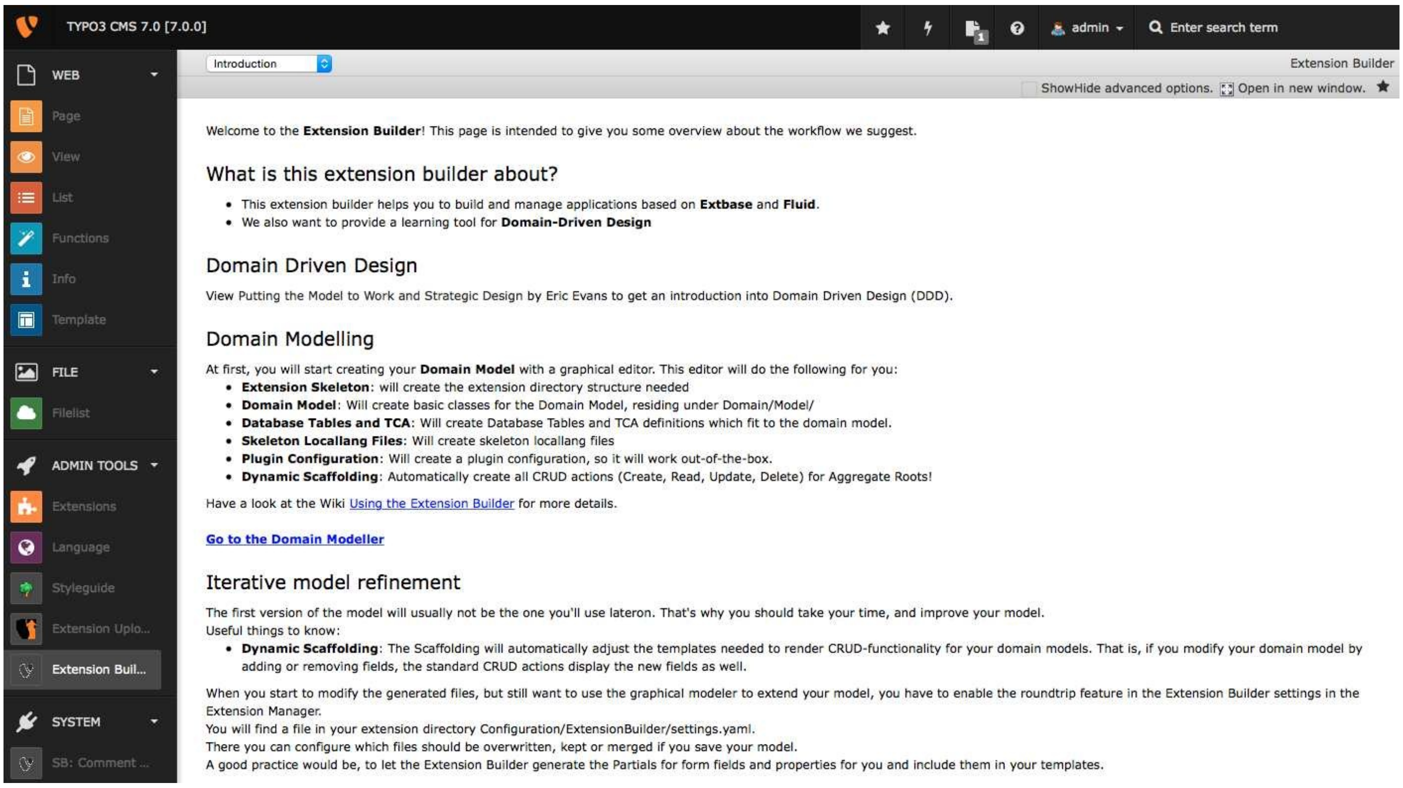
Installed Extensions

Search: Extension Builder

Upd.	A/D	Extension	Key	Version	Actions	State
	⊟	Extension Builder	extension_builder	7.0.0		beta

Following the installation, start the Extension Builder by clicking on its link in module *ADMIN TOOLS*.

Figure 5.3. Module Extension Builder



Welcome to the **Extension Builder**! This page is intended to give you some overview about the workflow we suggest.

What is this extension builder about?

- This extension builder helps you to build and manage applications based on **Extbase** and **Fluid**.
- We also want to provide a learning tool for **Domain-Driven Design**

Domain Driven Design

View Putting the Model to Work and Strategic Design by Eric Evans to get an introduction into Domain Driven Design (DDD).

Domain Modelling

At first, you will start creating your **Domain Model** with a graphical editor. This editor will do the following for you:

- **Extension Skeleton**: will create the extension directory structure needed
- **Domain Model**: Will create basic classes for the Domain Model, residing under Domain/Model/
- **Database Tables and TCA**: Will create Database Tables and TCA definitions which fit to the domain model.
- **Skeleton Locallang Files**: Will create skeleton locallang files
- **Plugin Configuration**: Will create a plugin configuration, so it will work out-of-the-box.
- **Dynamic Scaffolding**: Automatically create all CRUD actions (Create, Read, Update, Delete) for Aggregate Roots!

Have a look at the Wiki Using the Extension Builder for more details.

Go to the Domain Modeller

Iterative model refinement

The first version of the model will usually not be the one you'll use lateron. That's why you should take your time, and improve your model.
Useful things to know:

- **Dynamic Scaffolding**: The Scaffolding will automatically adjust the templates needed to render CRUD-functionality for your domain models. That is, if you modify your domain model by adding or removing fields, the standard CRUD actions display the new fields as well.

When you start to modify the generated files, but still want to use the graphical modeler to extend your model, you have to enable the roundtrip feature in the Extension Builder settings in the Extension Manager.
You will find a file in your extension directory Configuration/ExtensionBuilder/settings.yaml.
There you can configure which files should be overwritten, kept or merged if you save your model.
A good practice would be, to let the Extension Builder generate the Partials for form fields and properties for you and include them in your templates.

Select *Domain Modelling* from the dropdown box at the top.

Development snapshot of Extension Builder

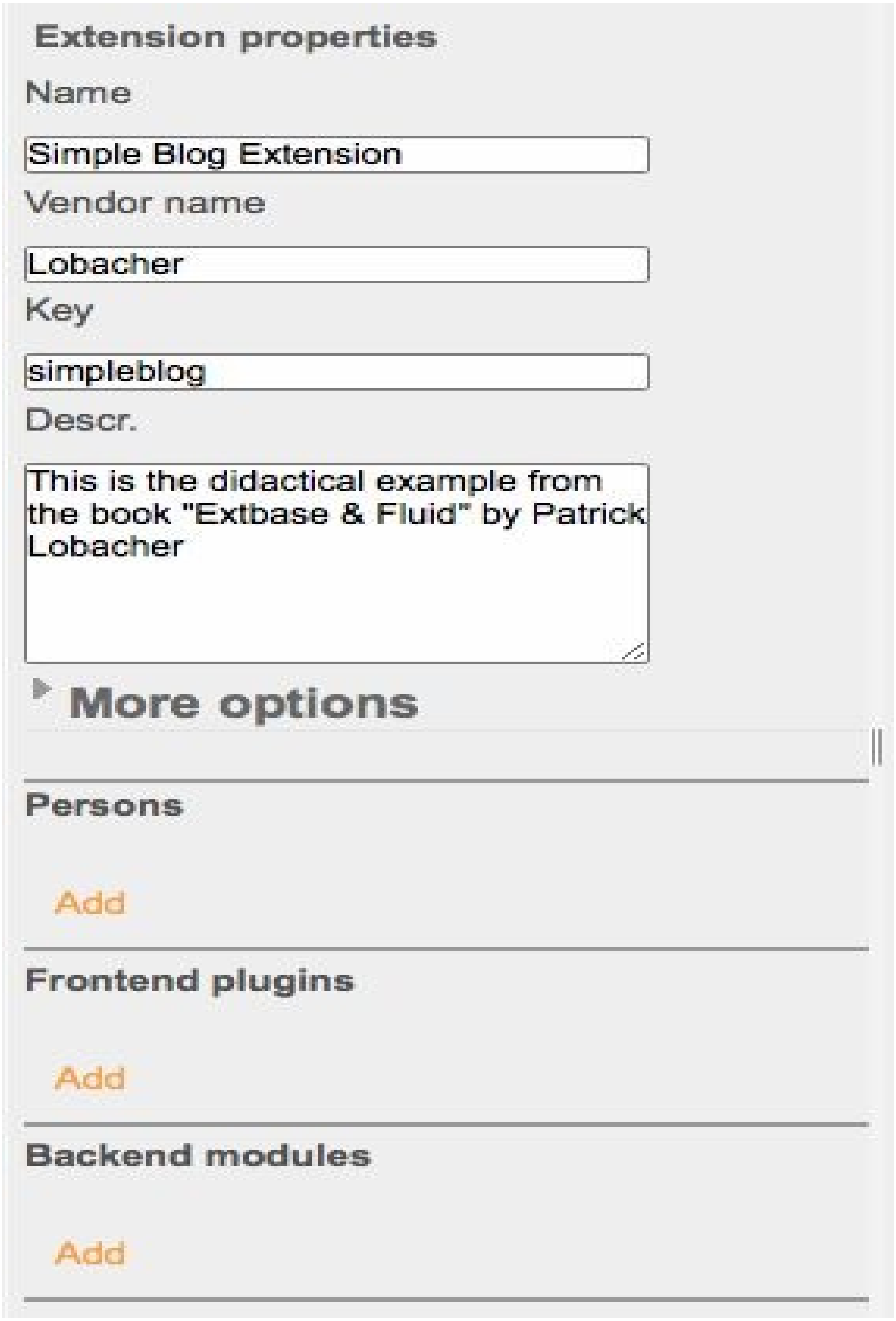
If you experience problems with the version of Extension Builder from TER or you would like to use the latest development version, you can also download the package from the source code repository.[14]

You can check-out the sources by using a Git client or download a snapshot from the website. Afterwards extract the archive, rename the directory to `extension_builder` and copy it to `typo3conf/ext/`. Now you should be able to install the extension as usual.

5.3.1. Extension Properties

Enter some details of your extension into the fields under *Extension properties* in the left column. This data is stored in file `ext_emconf.php` later. In order to ensure that the examples in the following sections of this book will work, its essential that you enter exactly the same values into the fields:

Figure 5.4. Extension properties, part 1



Extension properties

Name

Simple Blog Extension

Vendor name

Lobacher

Key

simpleblog

Descr.

This is the didactical example from the book "Extbase & Fluid" by Patrick Lobacher

Name

This is the name of the extension – enter `Simple Blog Extension` here.

Vendor name

This value is used as the first part of the package identifier (which consists of vendor name and extension key in lower case and without spaces) – enter `Lobacher` here, with the first letter in upper case.

Key

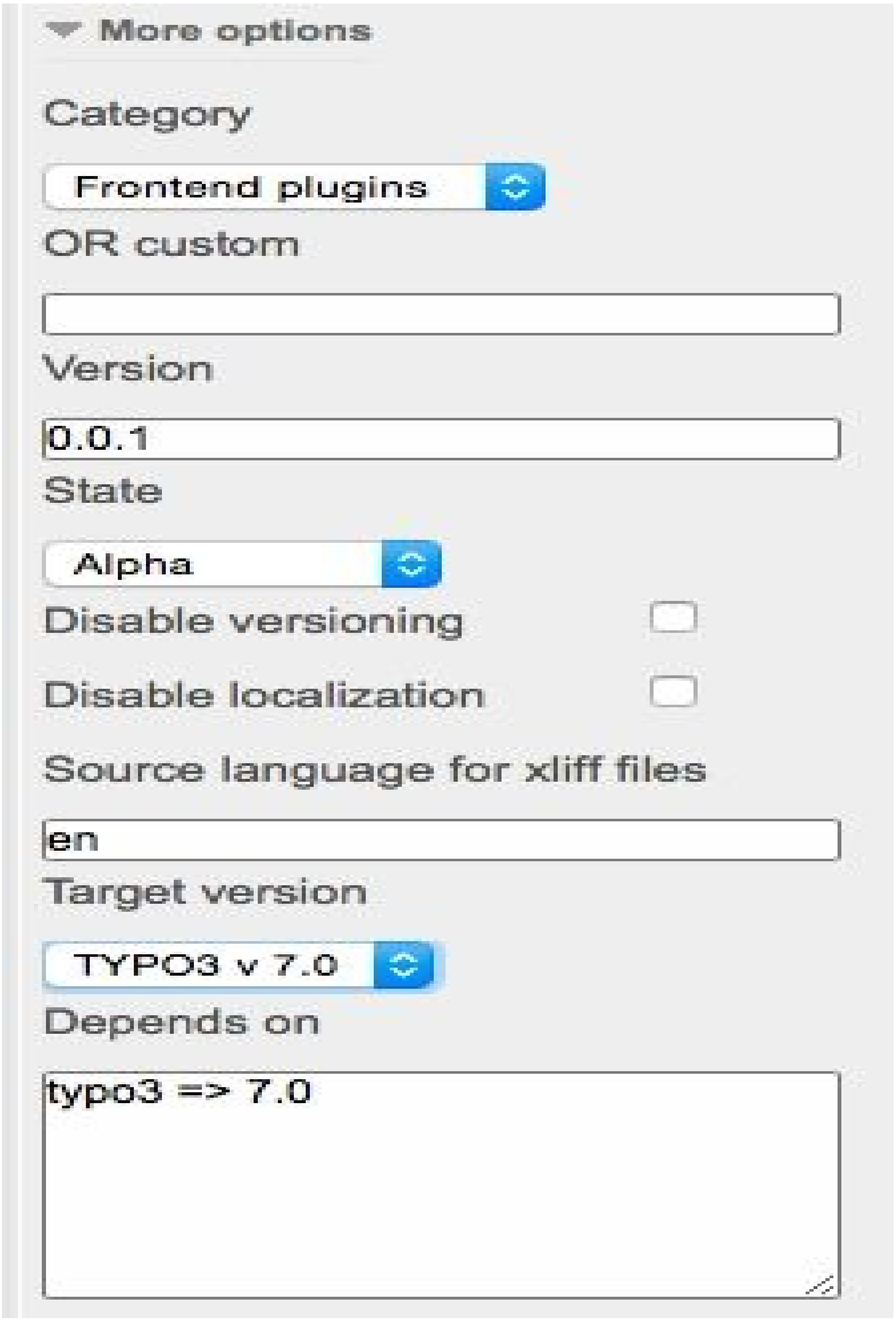
This is the extension key, which must be in lower case and without spaces – enter `simpleblog` here.

Description (Descr.)

This field expects the description of the extension as shown in the Extension Manager and in the TER (TYPO3 Extension Repository).

Now click on *More options*.

Figure 5.5. Extension properties, part 2



More options

Category

Frontend plugins

OR custom

Version

0.0.1

State

Alpha

Disable versioning ☐

Disable localization ☐

Source language for xliff files

en

Target version

TYPO3 v 7.0

Depends on

```
typo3 => 7.0
```

Category
: This configures the category – select `Frontend plugins` from the list.

OR custom
: It is also possible to define a custom category but in our case we will leave this field empty.

Version
: This option allows us to enter a version number in the format `major.minor.patchlevel` – we will start with 0.0.1.

State
: This is the current status of the extension. We just started with the development so select `Alpha`.

Disable versioning
: Activating this checkbox would skip the creation of versioning related columns in the database (these are: `t3ver_oid`, `t3ver_id`, `t3ver_wsid`, `t3ver_label`, `t3ver_state`, `t3ver_stage`, `t3ver_count`, `t3ver_tstamp`, `t3ver_move_id` and `t3_origuid`). We leave this checkbox disabled (unticked).

Disable localization
: Activating this checkbox would skip the creation of localisation related columns in the database.

Source language for xliff files
: The source language for xliff files is the language in which the *source* of a label is defined, which can be translated into other languages. Our selection is “en” for English.

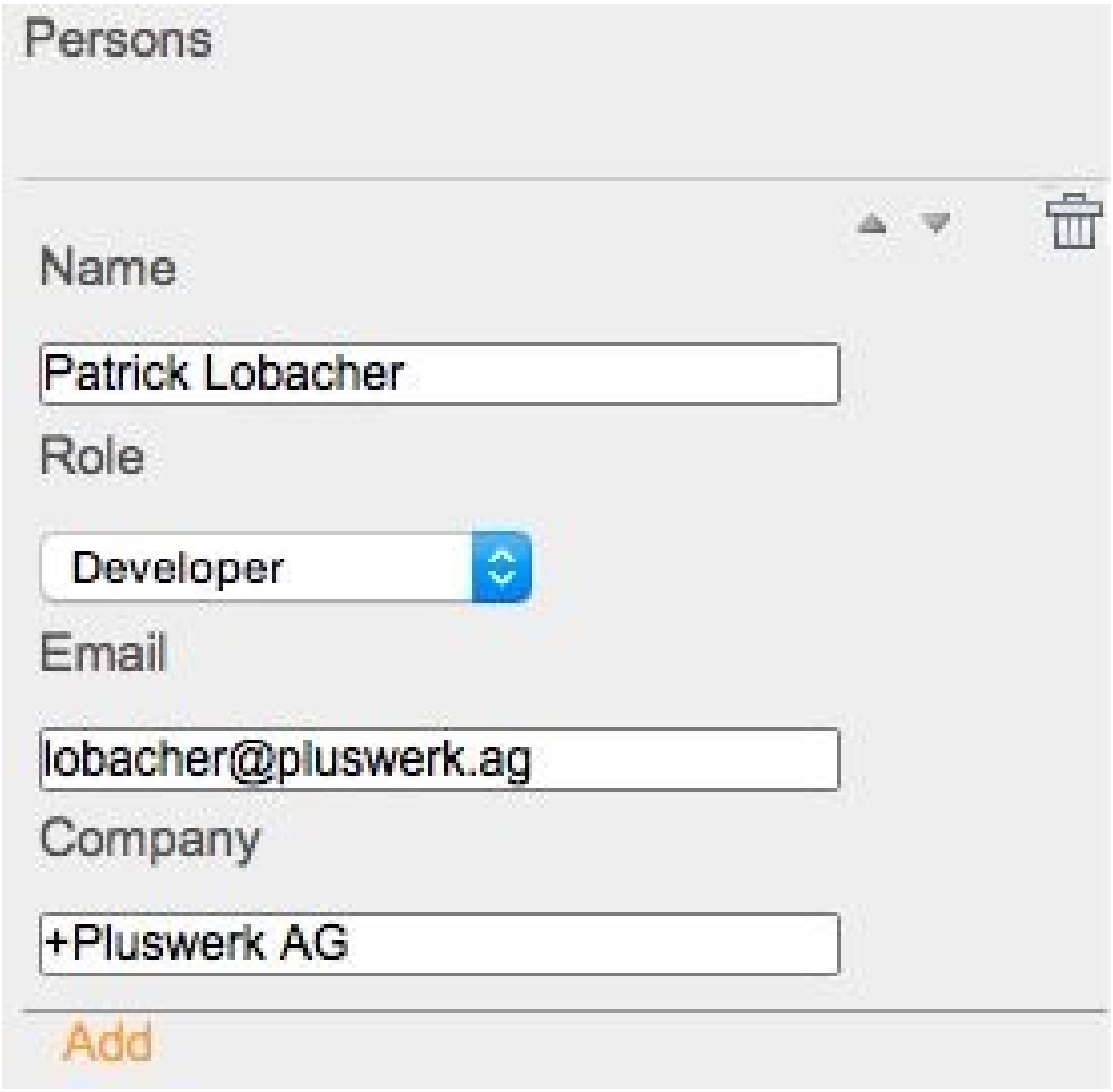
Target version
: Select the TYPO3 version the extension requires, for example `TYPO3 v 7.0`. If your extension requires a version that is not listed here, you can adjust this value manually in file `ext_emconf.php` later, once you save your configuration.

Depends on
: This textarea allows you to define if the extension depends on other extensions. These dependencies can be entered here, one per line (extension keys, followed by a double arrow and the lowest version number).

You should at least define the TYPO3 dependency because this is mandatory information for publishing your extension to the official TER.

Our next step is to click on *Add* under section *Persons*.

Figure 5.6. Extension properties, part 3


Persons

Name

Patrick Lobacher

Role

Developer

Email

lobacher@pluswerk.ag

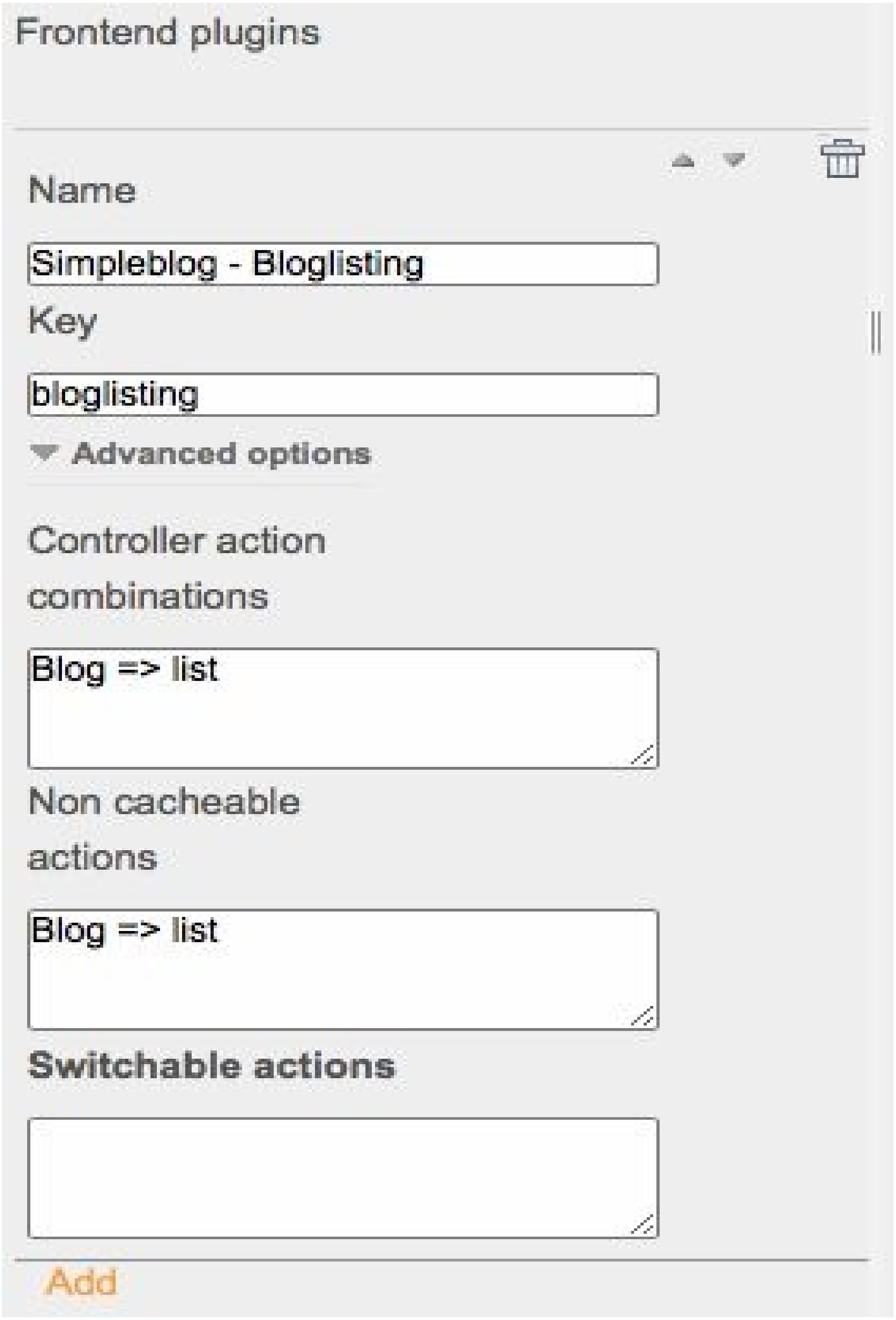
Company

+Pluswerk AG

These fields should be self-explanatory, except the dropdown box *Role*. According to the DDD principles, two options are available: `Developer` and `Product Manager` (client). Both can participate in the modelling process to create the domain model.

Afterwards, click on *Add* under section *Frontend plugins*.

Figure 5.7. Extension properties, part 4



Name: Simpleblog - Bloglisting

Key: bloglisting

Controller action combinations: Blog => list

Non cacheable actions: Blog => list

Switchable actions:

The purpose of this section is the configuration of the plugins of the extension. In principle, unlimited plugins may exist in an extension. This means that you can add as many plugins as you need.

Name

The value entered here will appear when a user adds the plugin as a content element to a page (list of items in the dropdown box). It is highly recommended to choose a meaningful name, which allows a user to determine what the functionality of the plugin is. We will use `Simpleblog - Bloglisting`.

Key

This input field excepts the unique name of the model, used as an identifier. It must not be re-used anywhere in the extension. Enter `Bloglisting` here.

Open the *Advanced options* by clicking the link.

Controller action combinations

Here you can configure, which controller action combinations the plugin should have (one controller per line, double arrow and the actions, comma separated). We are not sure which combinations will be required later so enter `Blog => list` in the interim.

Non cacheable actions

This textarea requires exactly the same notation as above. The difference is that the controller action combinations entered here will not have been cached. Enter `Blog => list` again.

Switchable actions

It is possible to make specific controller action combinations “switchable” so that some combinations are only legitimate under certain conditions. We will return to this later and leave this field empty for the time being.

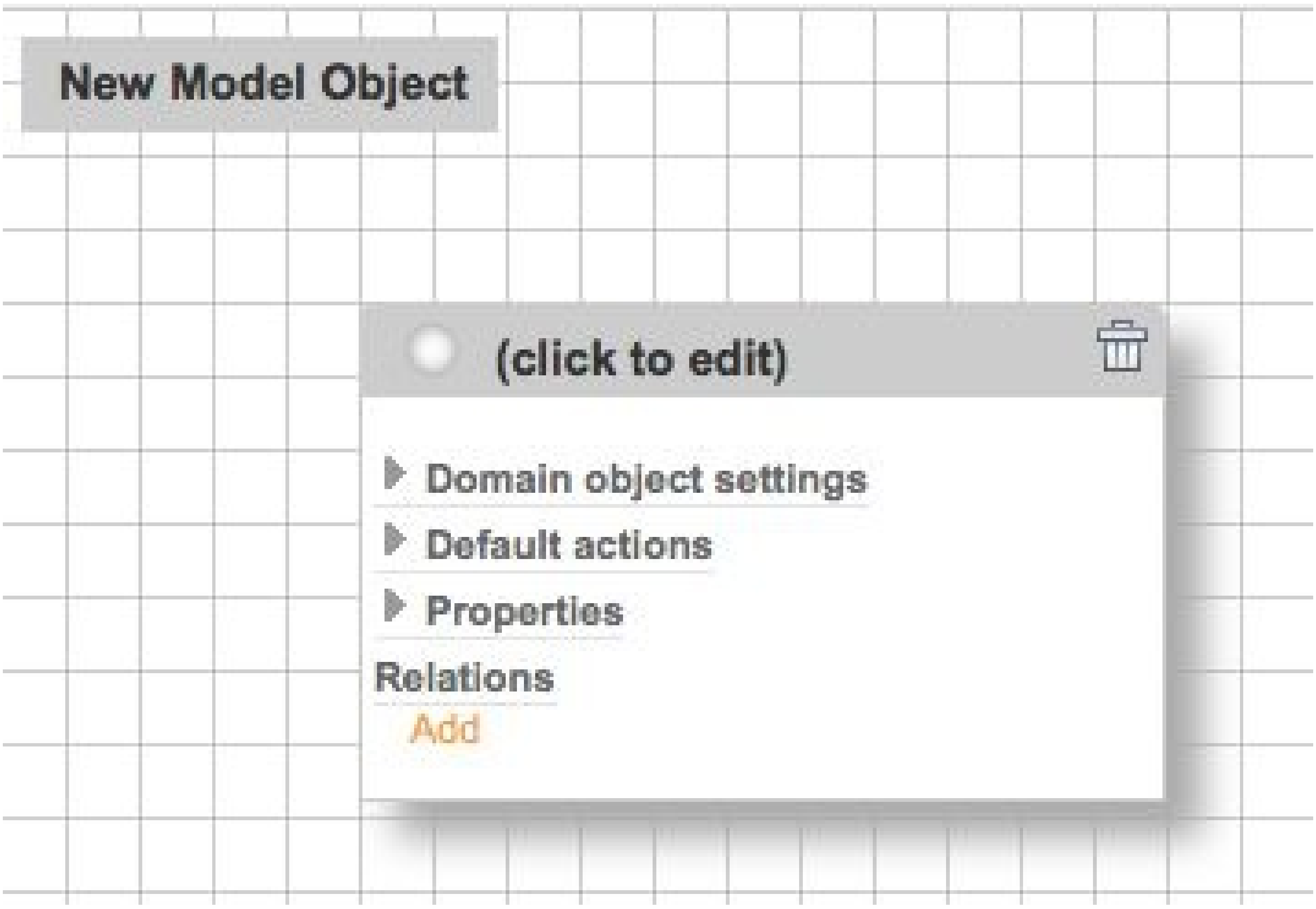
To avoid loosing the data just entered, we should save it now by clicking the *Save* button, centred at the bottom. Following the *Back* button of your browser or accessing any other module in TYPO3 or even if the backend session expires, then all this information will be gone. Therefore it is advised to store the data from time to time. Ignore possible warning/error messages by clicking *OK*.

5.3.2. Domain Model

At this point, the actual modelling begins. To gain some more space, we will hide the extension properties with the < icon.

To create a domain model, move your mouse over the *New Model Object* area, click and drag it into the grid area, where you can release the mouse button again (drag and drop).

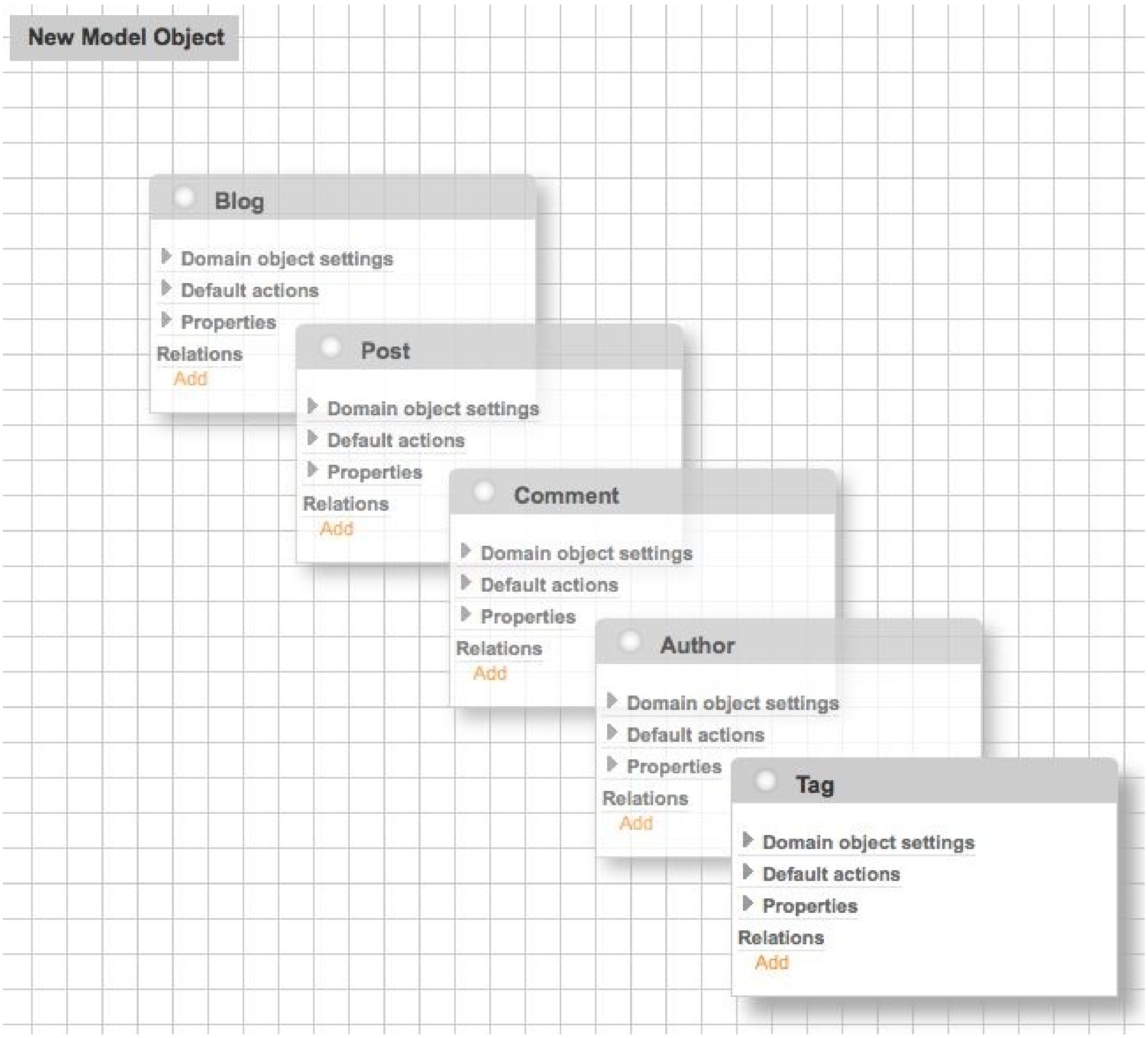
Figure 5.8. Drag and drop to create a new domain model



By clicking on *(click to edit)*, you can assign a name to the domain object. The name must be noted in UpperCamelCase. Enter `Blog` and continue with *Ok*.

Create four further domain objects by repeating this step (drag and drop from the “New Model Object” area) and assign the name of each object before continuing with the next one. Name names are: `Post`, `Comment`, `Author` and `Tag`.

Figure 5.9. Five domain objects created



The order of creating the objects matters

Keep in mind, that the order of how the objects are being created is important. The Extension Builder generates the code based on the order of the modelling. This might mean that something Post-related appears before Blog-related or similar. Technically speaking, this has no impact on the functionality but will possibly confuse developers, in particular beginners. After you are more confident with Extbase, this does not matter any more.

5.3.3. Domain Model Properties

In this step we enter all properties at the appropriate objects – in accordance with the domain model.

Properties of the Domaim Object “Blog”

Click *Domain object settings* and define the properties as follows:

Object type
: This allows us to configure the object as an `Entity` or `Value Object`. In accordance with the Domain Model, we choose `Entity`.

Is aggregate root
: Due to the fact that the object is also an `Aggregate Root`, this checkbox needs to be ticked.

Enable sorting
: If the data records should be manually sortable in the backend later, this can be activated here. This is not required in our case so we will leave the checkbox deactivated.

Add deleted field
: Adds a field `deleted` in the database to indicate that the record was deleted. This checkbox needs to be ticked.

Add hidden field
: Adds a field `hidden` in the database to indicate that the record is hidden. This checkbox needs to be ticked.

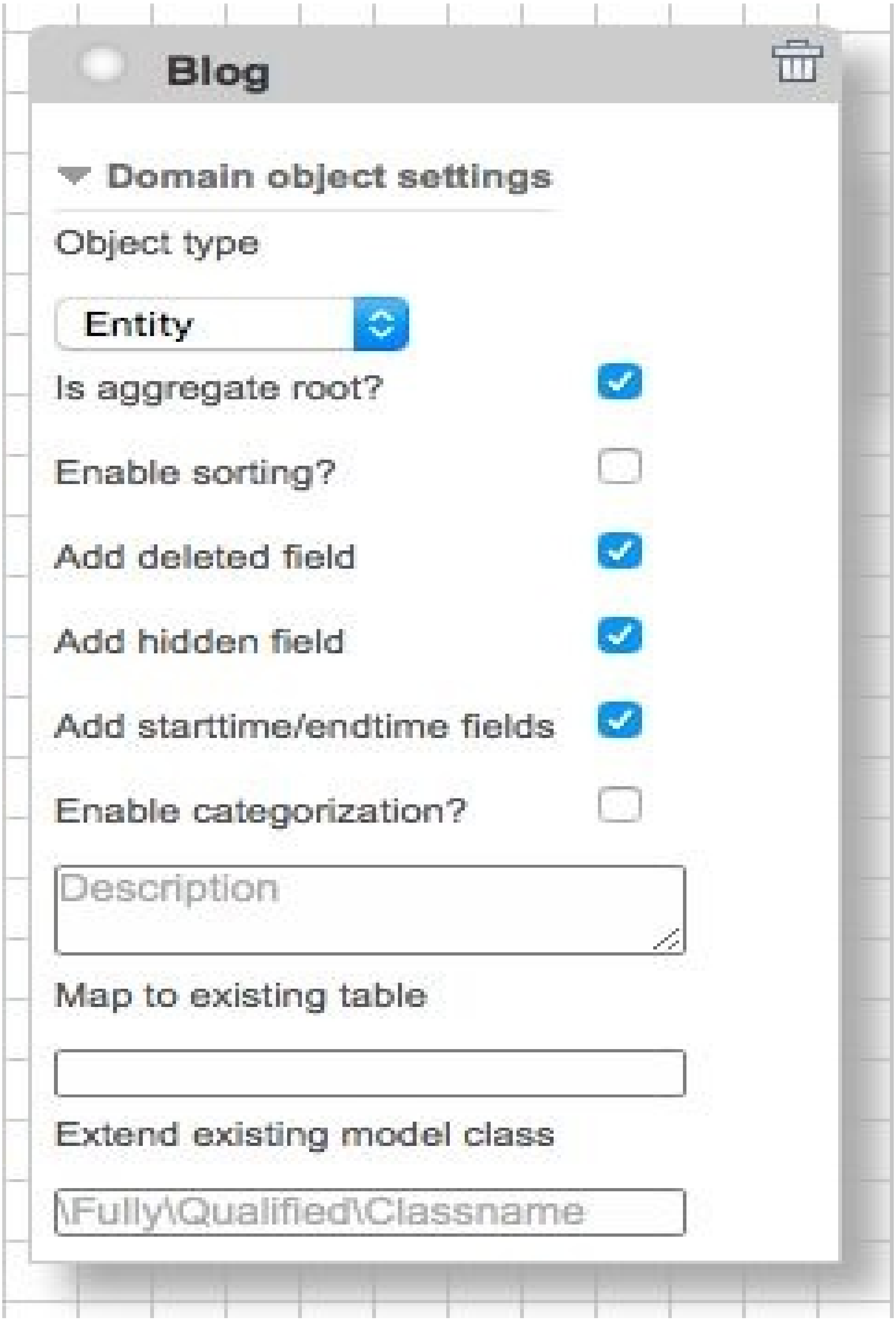
Enable categorization?
: If categories should be added to the record later, this can be activated here. This is not required in our case so we will leave the checkbox deactivated.

Description
: This input field can be used to store a short description, which is also visible in the backend’s list view as the column label. Therefore there are two options: you could either choose `Blogs` (in this case the column labels match the records) or you choose something more meaningful to describe the domain object – and possibly amend the labels manually later.

Map to existing table
: It is possible to map a domain object to an existing database table as a basic principle. To achieve this, the table name must be entered here. Although we will use the functionality behind this feature at the domain object `Author` (by mapping the object to the table `fe_users`), we will leave this field empty for all objects.

Extend existing model class
: If this model class should be deviated from another model class, this could be configured in this field. We will leave this empty.

Figure 5.10. Domain object settings of domain object “Blog”



Blog

Domain object settings

Object type

Entity

Is aggregate root? ✓

Enable sorting? ☐

Add deleted field ✓

Add hidden field ✓

Add starttime/endtime fields ✓

Enable categorization? ☐

Description

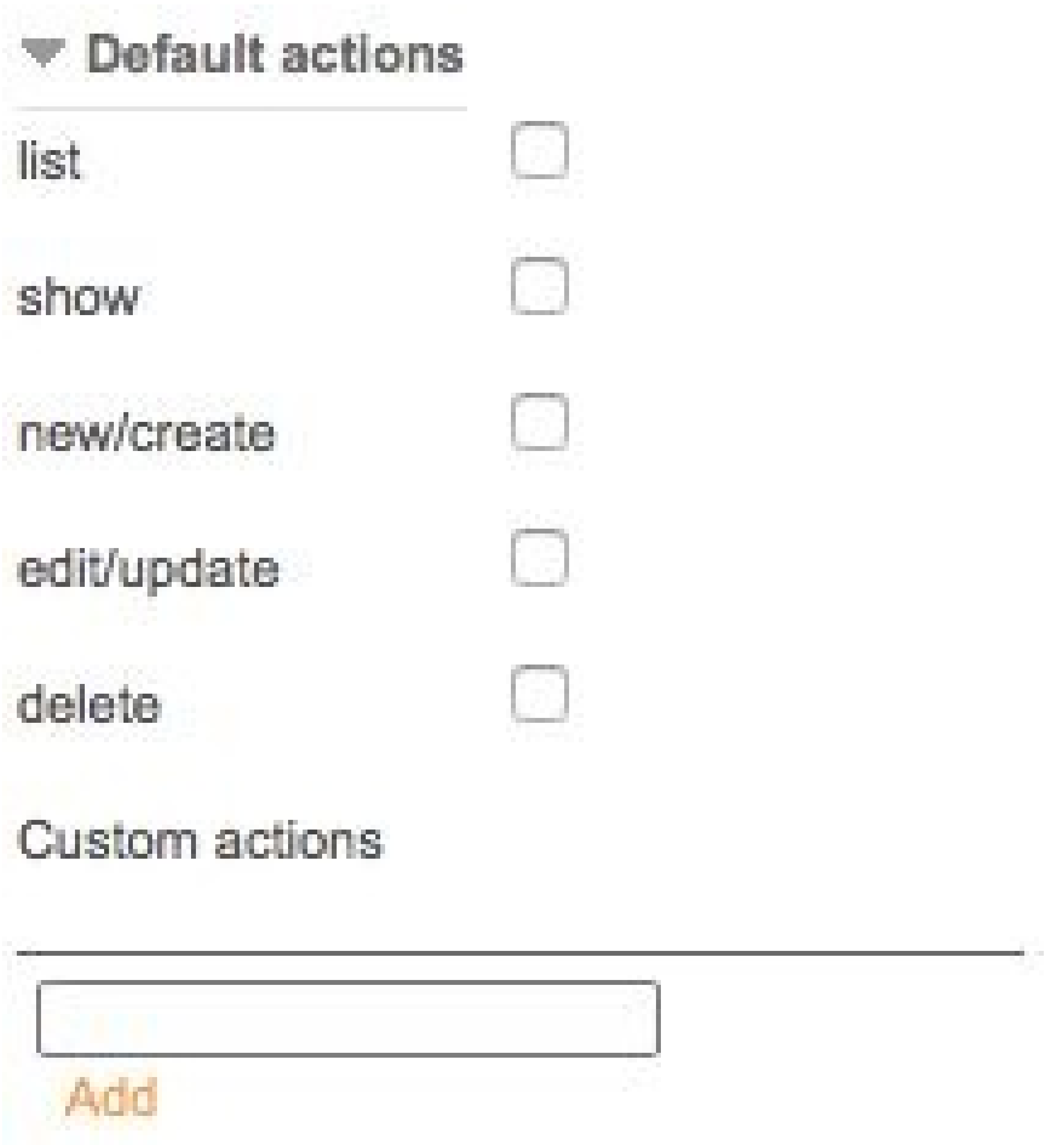
Map to existing table

Extend existing model class

\Fully\Qualified\Classname

Continue with the *Default Actions*: This section allows us to select *Default Actions*, which are code fragments provided by the Extension Builder (or custom actions). However we want to work out which where these Actions exist and when they are triggered so we can leave this configuration empty for all domain objects.

Figure 5.11. Default actions remain empty



Next, the properties are created. Click on the link *Properties* and then *Add*.

Enter the following details for the property title:

Property name

The name of the property in lowerCamelCase (without any special characters and without underscore). Enter `title` here.

Property type

Select the type of the property from the dropdown box. This affects the display in the backend (by the underlying Table Configuration Array – TCA). An overview of available types can be found in the appendix. For our property, we choose `String`.

Description

Enter a short description of the property into this input field.

Is required

Two things happen from a technical perspective if you activate this checkbox. The TCA is being configured to store this information only, if the value is not empty. Secondly, a *NotEmpty-Validator* is added to the annotation of the property. We will come back to both of these actions in detail later. At this time, we activate the checkbox.

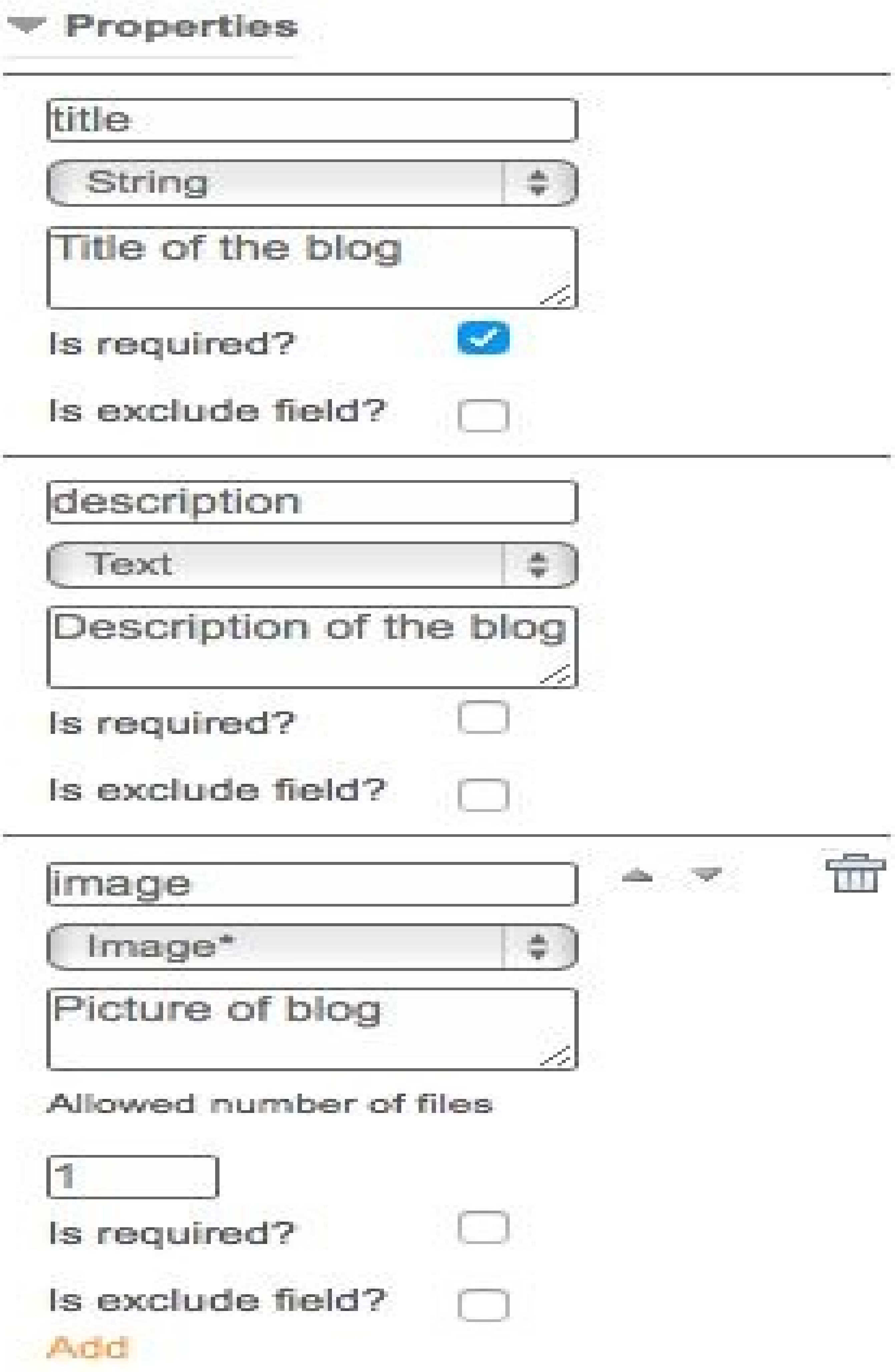
Is excluded field

In TYPO3 CMS it is possible to configure user groups and deny access to specific fields for them. These fields will not be visible to corresponding users in the backend. In order to enable the feature to hide field, it has to be defined as an `ExcludeField` in the TCA. If you enable this checkbox it will be possible to hide the field later. However this makes no sense for required fields so we can leave this checkbox disabled.

Keep in mind that all fields can be manually amended later, no matter what we have chosen here at this point in time.

The remaining properties `description` and `image` should be configured as shown in Figure 5.12.

Figure 5.12. Properties of the domain object “Blog”



Properties

title

String

Title of the blog

Is required? [✓]

Is exclude field? ☐

description

Text

Description of the blog

Is required? ☐

Is exclude field? ☐

image

Image*

Picture of blog

Allowed number of files

1

Is required? ☐

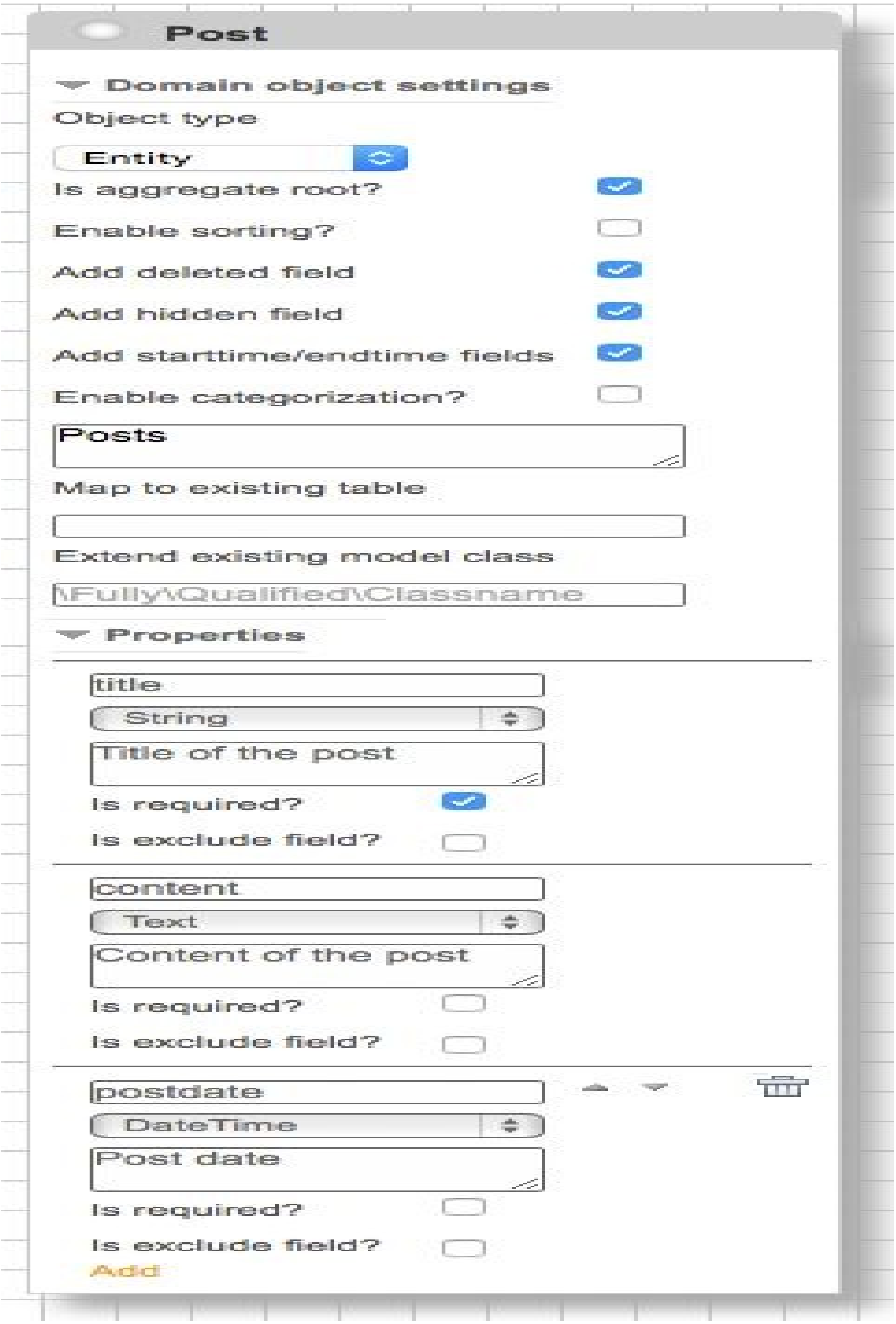
Is exclude field? ☐

Add

Properties of the Domain Object “Post”

Complete the form of the object Post as shown in Figure 5.13.

Figure 5.13. Properties of the domain object “Post”



Post

Domain object settings

Object type

Entity

Is aggregate root? ✓

Enable sorting? ☐

Add deleted field ✓

Add hidden field ✓

Add starttime/endtime fields ✓

Enable categorization? ☐

Posts

Map to existing table

Extend existing model class

\Fully\Qualified\Classname

Properties

title

String

Title of the post

Is required? ✓

Is exclude field? ☐

content

Text

Content of the post

Is required? ☐

Is exclude field? ☐

postdate

DateTime

Post date

Is required? ☐

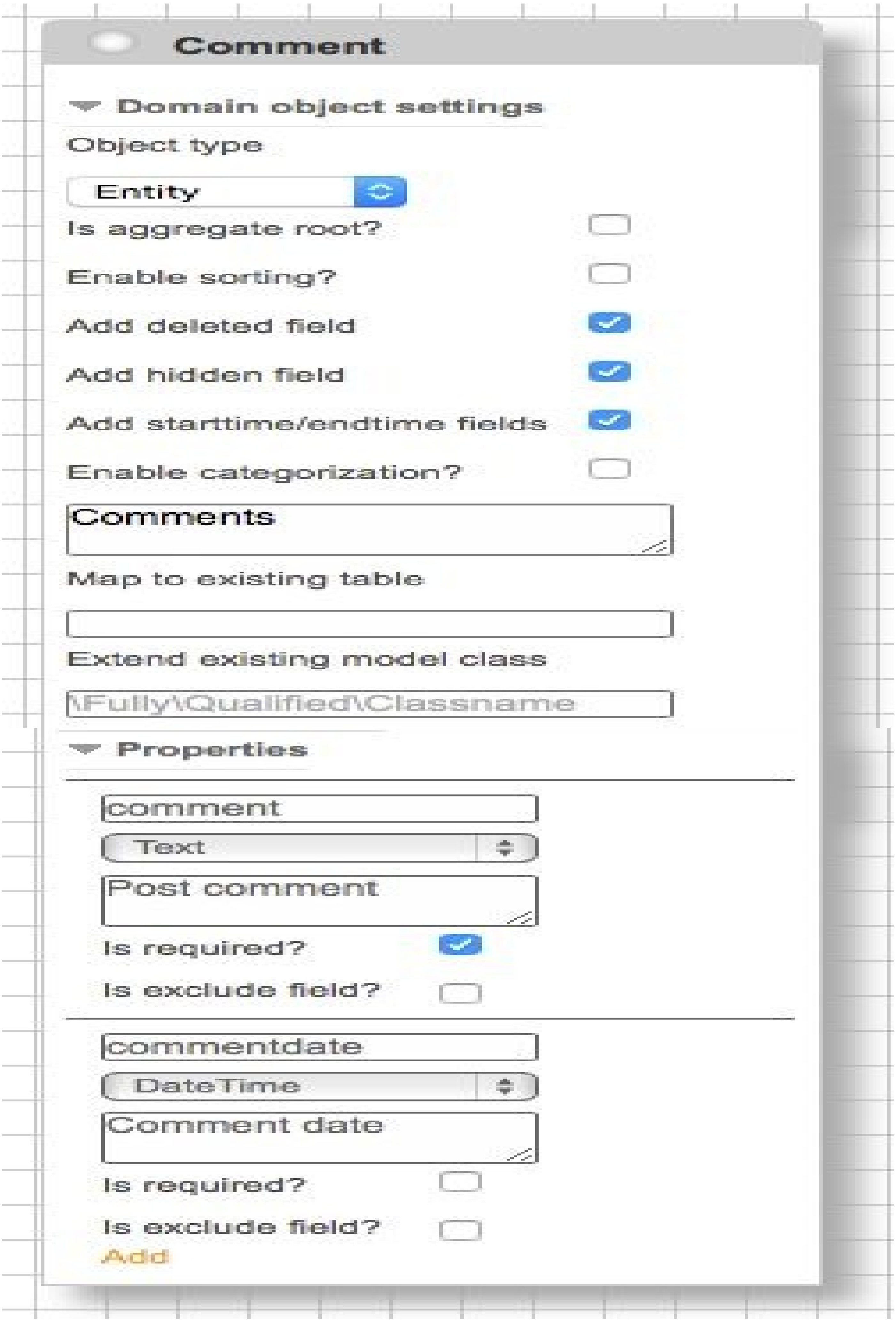
Is exclude field? ☐

Add

Properties of the Domain Object “Comment”

The same for the object `Comment` but note that this object is not an `Aggregate Root`.

Figure 5.14. Properties of the Domain Object “Comment”]



Comment

Domain object settings

Object type

Entity

Is aggregate root? ☐

Enable sorting? ☐

Add deleted field ✓

Add hidden field ✓

Add starttime/endtime fields ✓

Enable categorization? ☐

Comments

Map to existing table

Extend existing model class

\Fully\Qualified\Classname

Properties

comment

Text

Post comment

Is required? ✓

Is exclude field? ☐

commentdate

DateTime

Comment date

Is required? ☐

Is exclude field? ☐

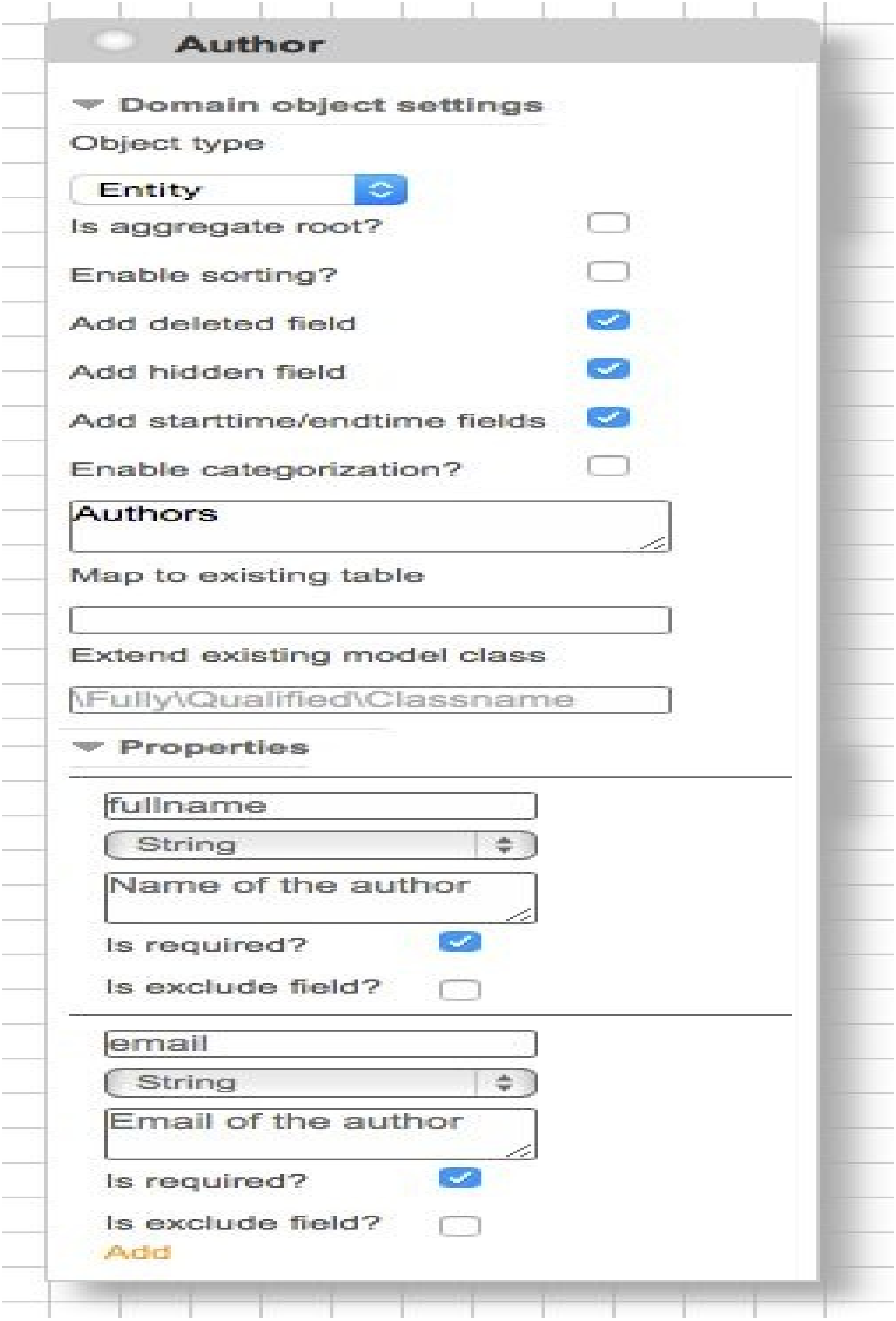
Add

Field `commentdate` should not be defined as required because we will populate its values programmatically.

Properties of the Domain Object “Author”

Continue with the object `Author`. This object is not an `Aggregate Root` either.

Figure 5.15. Properties of the Domain Object “Author”



Author

Domain object settings

Object type

Entity

Is aggregate root? ☐

Enable sorting? ☐

Add deleted field ✓

Add hidden field ✓

Add starttime/endtime fields ✓

Enable categorization? ☐

Authors

Map to existing table

Extend existing model class

\Fully\Qualified\Classname

Properties

fullname

String

Name of the author

Is required? ✓

Is exclude field? ☐

email

String

Email of the author

Is required? ✓

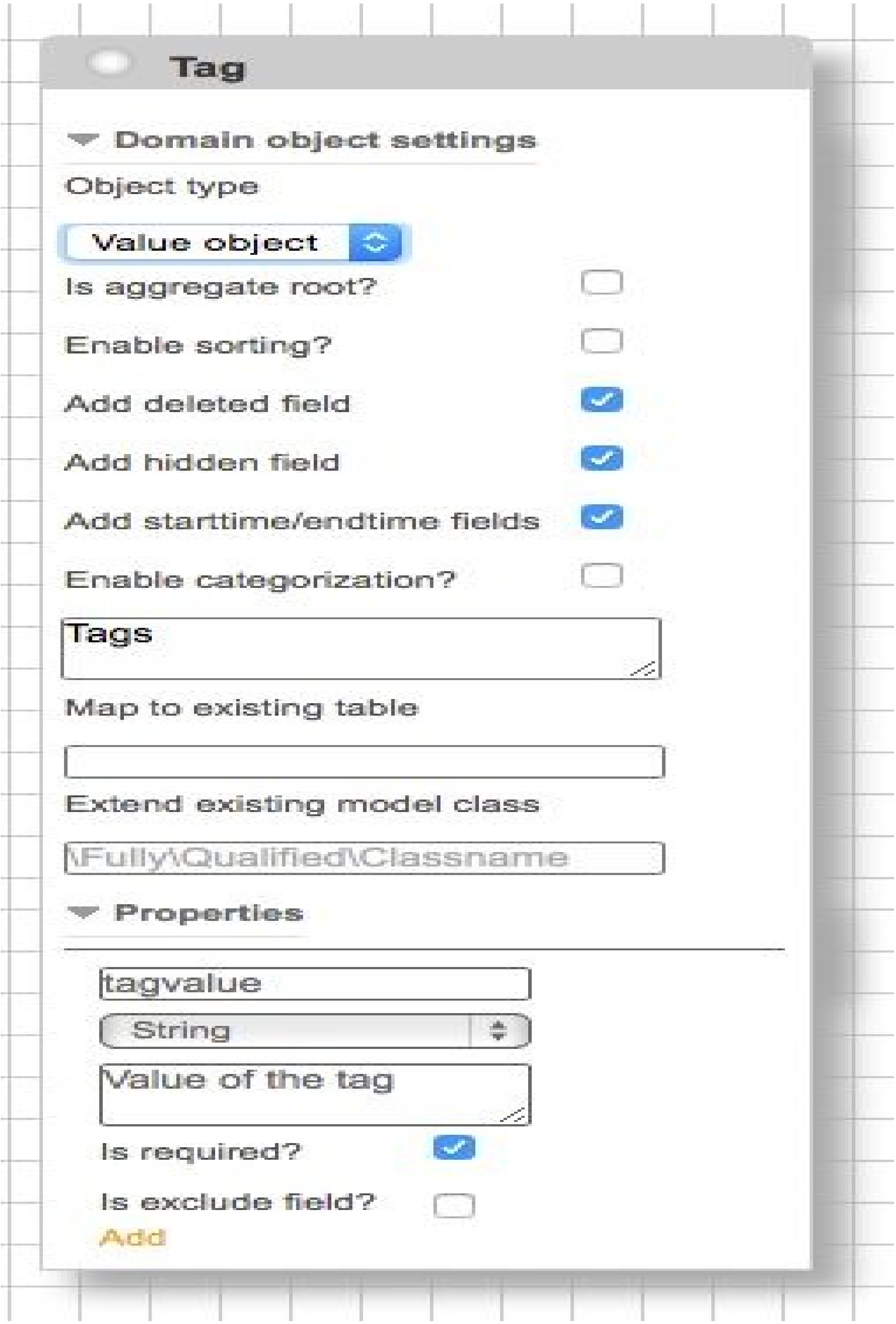
Is exclude field? ☐

Add

Properties of the domain object “Tag”

Finally, the object `Tag`. The important fact with this object is, that the object type is `Value Object`.

Figure 5.16. Properties of the domain object “Tag”



Tag

Domain object settings

Object type

Value object

Is aggregate root? ☐

Enable sorting? ☐

Add deleted field ✓

Add hidden field ✓

Add starttime/endtime fields ✓

Enable categorization? ☐

Tags

Map to existing table

Extend existing model class

\Fully\Qualified\Classname

Properties

tagvalue

String

Value of the tag

Is required? ✓

Is exclude field? ☐

Add

You should save the domain model now.

5.3.4. Domain Model Relations

Now it is time to model the relations between domain objects.

5.3.5. Relation Between Blog and Post

Close all forms of all objects and position object “Blog` at the left hand side and object `Post` at the right hand side in juxtaposition with each other. Open the Relations” properties by clicking *Add* at the bottom (below Relations) and also open advances properties (link *More*).

Name
: Enter the name of the relation – in lowerCamelCase again. This is ultimately a property of the object. Since the relation (by 1:n) may contain unlimited posts, we will name the property `posts`.

Type
: The type of the relation (`1:1`, `1:n`, `n:1` or `m:n`) is to be defined here. We choose `1:n`.

Description
: This fields hold a short description.

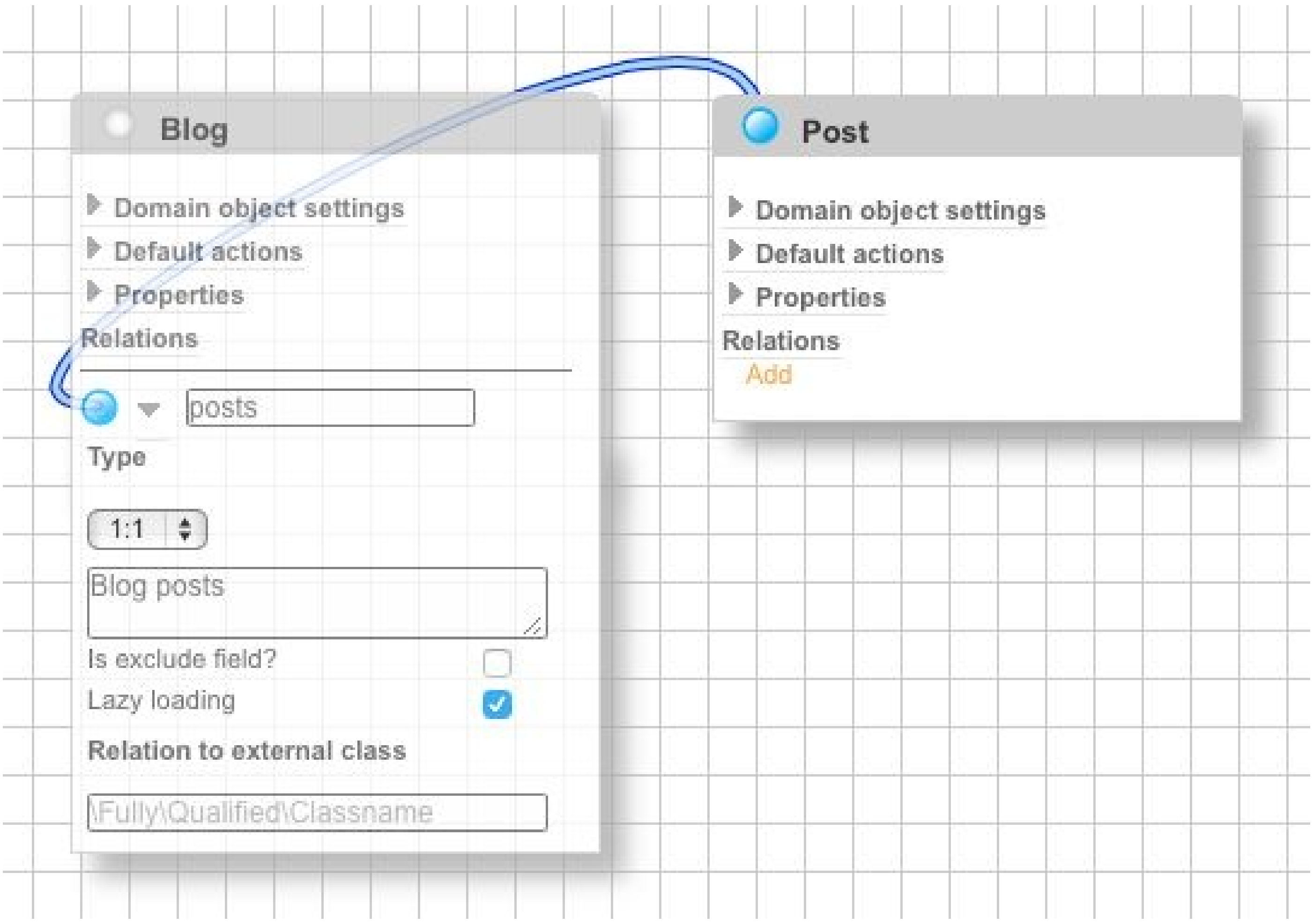
Is exclude field
: The same functionality as the field with the name under domain model properties applies here. We will leave this field empty.

Lazy Loading
: Usually objects are loaded by Extbase including their child objects. This means for a Blog with 1000 posts, that all posts are read immediately. In most cases, this is not desirable. By activating the `Lazy Loading` option, reading the data is postponed to the time, when the property `posts` is really required by Extbase. Activate this option.

Relation to external class
: At this point you can define a (fully qualified) class name, if the relation should be connected to an external model class. In our case, we leave this field empty.

Now, move the mouse over the grey circle, left of the relation name, click, drag and drop the blue “snail” (*relation wire*) to the grey circle of the domain model `Post` and release the mouse button. The relation has been created.

Figure 5.17. Relation “posts” between Blog and Post



posts

Type

1:1

Blog posts

Is exclude field? ☐

Lazy loading ✓

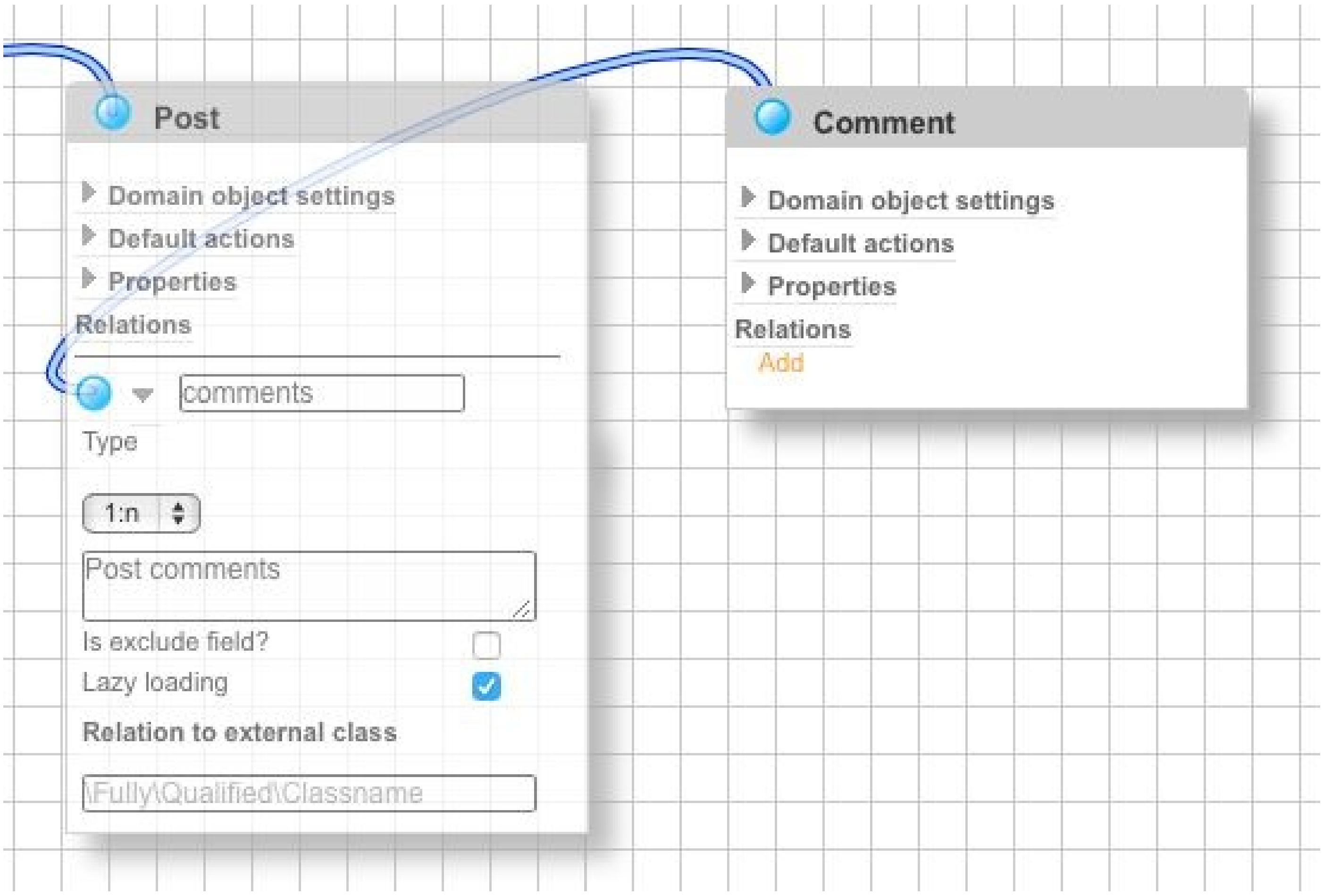
Relation to external class

\Fully\Qualified\Classname

5.3.6. Relation Between “Post” and “Comment”

By using the same method, create the relation between `Post` and `Comment` and name it `comments`.

Figure 5.18. Relation “comments” between “Posts” und “Comments”

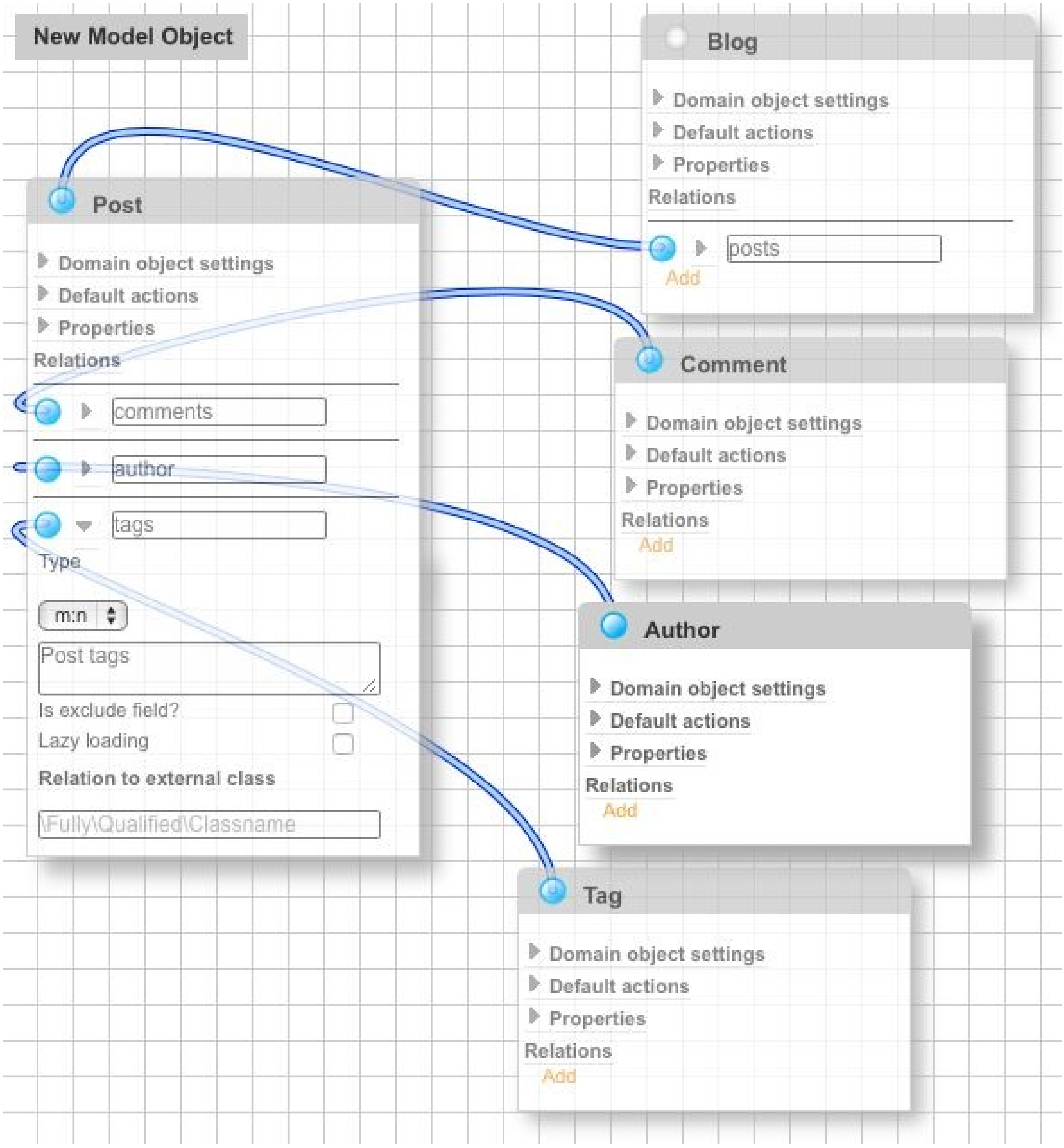


comments
Type
1:n
Post comments
Is exclude field? ☐
Lazy loading ✓
Relation to external class
\Fully\Qualified\Classname

5.3.7. Relations Between “Post” and “Author”/“Tag”

The domain object `Post` has two additional relations with objects `Author` and `Tag` – all values can be determined from the model.

Figure 5.19. Overview of all relations

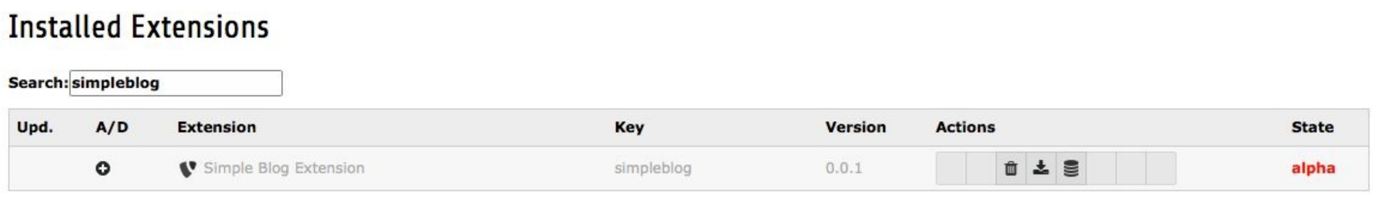


Now is the perfect time to save your work.

5.4. Installation of the Extension

When you save the work in the Extension Builder, the extension is created but not installed. Open the TYPO3's Extension Manager and install the extension.

Figure 5.20. Installation of extension "simpleblog"



Installed Extensions

Search: simpleblog

Upd.	A/D	Extension	Key	Version	Actions	State
	⊕	Simple Blog Extension	simpleblog	0.0.1		alpha

Afterwards, add the plugin to a page of your choice. To achieve this, select *new content element* and switch to tab *Plugins*. Select *Simpleblog – Bloglisting* from the list of available plugins under *General Plugin*.

If you use a pre-configured system, you will already see an output (in this case an error message). In the case you are using an empty system, you have to create a TypoScript template.

This can be done by opening module *Template* in the backend, selecting the function *Info/Modify* and clicking the button *Create template for a new site*. Edit the TypoScript template (e.g. follow link *Edit the whole template record*) and enter the following code in the setup:

```
page = PAGE
page.10 < styles.content.get
```

Now switch to tab *Includes* and choose *CSS Styled Content (css_styled_content)* from *Available Items* (right) under the *Include static (from extensions)* section. The item appears in the left field. Finally, save the template record by using the *Save and close document* icon at the top (disk with "X").

In the next chapter we will analyse all files created and extend the extension by the functionality required.

5.5. Analysing Files Created by Extension Builder

The Extension Builder generates a number of files, which we will review in this chapter.

`Classes`
: All class files of an extension must be stored in this folder and all files in this folder must be class files.

`Classes/Controller`
: All controllers of the extension.

`Classes/Controller/BlogController.php`
: Controller of domain object `Blog`.

`Classes/Controller/PostController.php`
: Controller of domain object `Post`.

`Domain`
: All domain related files are stored in this folder.

`Domain/Model`
: Domain object classes.

`Domain/Model/Author.php`
: Domain class of objecr `Author`

`Domain/Model/Blog.php`
: Domain class of object `Blog`

`Domain/Model/Comment.php`
: Domain class of object `Comment`

`Domain/Model/Post.php`
: Domain class of object `Post`

`Domain/Model/Tag.php`
: Domain class of object `Tag`

`Domain/Repository`
: All Repository classes are stored in this folder.

`Domain/Repository/BlogRepository.php`
: Repository class of domain object `Blog`

`Domain/Repository/PostRepository.php`
: Repository class of domain object `Post`

`Configuration`
: All configuration files are stored in this folder.

`Configuration/ExtensionBuilder`
: Configuration files of the Extension Builder are stored in this folder.

`Configuration/ExtensionBuilder/settings.yaml`
: Settings of the Extension Builder, which allows for the configuration of how files are created.

`Configuration/TCA`
: All TCA files are stored in this folder.

`Configuration/TCA/Author.php`
: TCA configuration of domain object `Author`

`Configuration/TCA/Blog.php`
: TCA configuration of domain object `Blog`

`Configuration/TCA/Comment.php`
: TCA configuration of domain object `Comment`

`Configuration/TCA/Post.php`
: TCA configuration of domain object `Post`

`Configuration/TCA/Tag.php`
: TCA configuration of domain object `Tag`

`Configuration/TypoScript`
: Static TypoScript of the extension is stored in this folder.

`Configuration/TypoScript/constants.txt`
: TypoScript constants.

`Configuration/TypoScript/setup.txt`
: TypoScript setup.

`Documentation.tmpl`
: A template of the manual as a set of reStructured Text files (ReST).

`Documentation.tmpl/Index.rst`
: Raw example main file of the manual which includes other rst files from this folder and from sub-folders.

`Resources`
: Resource files – both private as well as public are stored in this folder.

`Resources/Private`
: Private resource files: these are files, which need post-processing, such as language files, templates, etc.

`Resources/Private/Language`
: Language files are stored in this folder.

`Resources/Private/Language/locallang.xlf`
: Main language file for the frontend. This files contains all labels, which have been entered in the Modeller.

`Resources/Private/Language/locallang_csh_tx_simpleblog_domain_model_author.x`
: Language file for the context sensitive help (CSH) in the backend of object `Author`

`Resources/Private/Language/locallang_csh_tx_simpleblog_domain_model_blog.xlf`
: Language file for the context sensitive help (CSH) in the backend of object `Blog`

`Resources/Private/Language/locallang_csh_tx_simpleblog_domain_model_comment.xlf`
: Language file for the context sensitive help (CSH) in the backend of object `Comment`

`Resources/Private/Language/locallang_csh_tx_simpleblog_domain_model_post.xlf`
: Language file for the context sensitive help (CSH) in the backend of object `Post`

`Resources/Private/Language/locallang_csh_tx_simpleblog_domain_model_tag.xlf`
: Language file for the context sensitive help (CSH) in the backend of object `Tag`

`Resources/Private/Language/locallang_db.xlf`
: Language file for all database-related labels in the backend.

`Resources/Public`
: Public resource files: these are files, which can be loaded/shown in the frontend or backend directly and without further processing. For example CSS files, JavaScript files, icons, etc.

`Resources/Public/Icons`
: All icons used.

`Resources/Public/Icons/relation.gif`
: Icon of a relation (used in the backend).

`Resources/Public/Icons/tx_simpleblog_domain_model_author.gif`
: Icon of object `Author` (used in the backend).

`Resources/Public/Icons/tx_simpleblog_domain_model_blog.gif`
: Icon of object `Blog` (used in the backend).

`Resources/Public/Icons/tx_simpleblog_domain_model_comment.gif`
: Icon of object `Comment` (used in the backend).

`Resources/Public/Icons/tx_simpleblog_domain_model_post.gif`
: Icon of object `Post` (used in the backend).

`Resources/Public/Icons/tx_simpleblog_domain_model_tag.gif`
: Icon of object `Tag` (used in the backend).

`Tests`
: Files for automated tests are stored in this folder.

`Tests/Unit`
: Unit tests.

`Tests/Unit/Controller`
: Unit test files of the controller are stored in this folder.

`Tests/Unit/Controller/BlogController.php`
: Unit test file of the controller `Blog`.

`Tests/Unit/Controller/PostController.php`
: Unit test file of the controller `Post`.

`Tests/Unit/Domain`
: Unit test files of the domain are stored in this folder.

`Tests/Unit/Domain/Model`
: Unit test files of the model are stored in this folder.

`Tests/Unit/Domain/Model/AuthorTest.php`
: Unit test files of the model of object `Author`

`Tests/Unit/Domain/Model/BlogTest.php`
: Unit test files of the model of object `Blog`

`Tests/Unit/Domain/Model/CommentTest.php`
: Unit test files of the model of object `Comment`

`Tests/Unit/Domain/Model/PostTest.php`
: Unit test files of the model of object `Post`

`Tests/Unit/Domain/Model/TagTest.php`
: Unit test files of the model of object `Tag`

`ext_emconf.php`
: Extension configuration file.

`ext_icon.gif`
: Icon of the extension, which will be shown in TER and TYPO3's Extension Manager.

`ext_localconf.php`
: Configuration file for the frontend.

`ext_tables.php`
: Configuration file for the backend.

`ext_tables.sql`
: SQL file to create database tables required.

`ExtensionBuilder.json`
: This configuration file contains all information entered in the Extension Builder, even the positions of the domain objects in the Modeller. Such a file is required to re-open the extension in the Modeller at a later time.

5.6. Further Functions of the Extension Builder

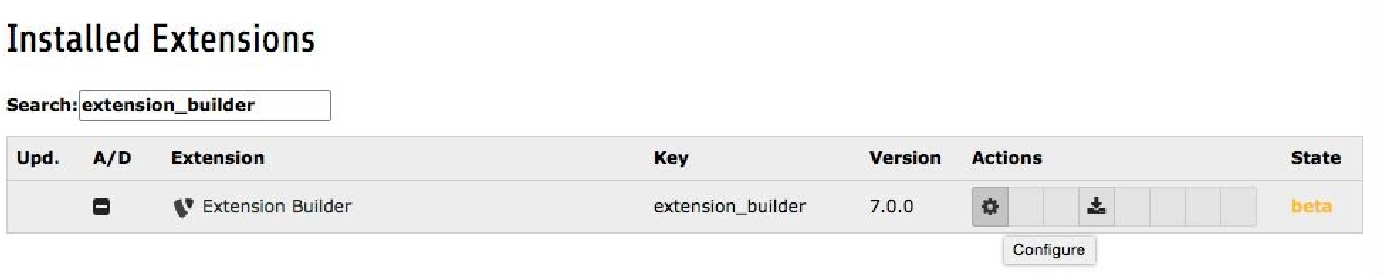
We learnt in Chapter 3 that changes of the domain model may occur during the project. DDD requires the option to adjust the implementation accordingly.

The Extension Builder supports this without problems.

5.6.1. Create a Backup

Go to the Extension Manager, select *Manage Extensions* and locate the extension `extension_builder`. Move your mouse over the *Actions* area and click on the appearing gear-wheel to configure the extension.

Figure 5.21. Extension settings in Extension Manager



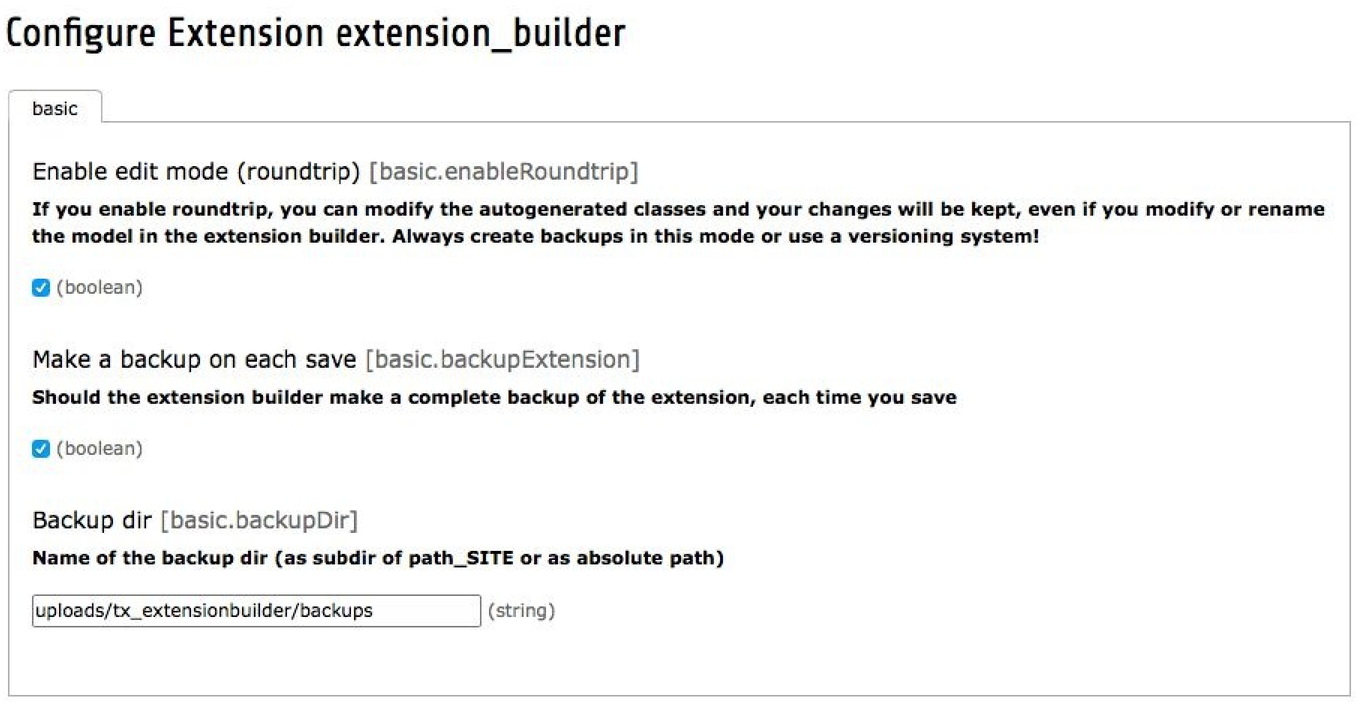
Installed Extensions

Search: extension_builder

Upd.	A/D	Extension	Key	Version	Actions	State
	⊟	Extension Builder	extension_builder	7.0.0	⚙ ⤓	beta

Activate all checkboxes which are shown on the settings page and save your new configuration by clicking on the disk icon at the top.

Figure 5.22. Extension Builder configuration



Configure Extension extension_builder

Enable edit mode (roundtrip) [basic.enableRoundtrip]

If you enable roundtrip, you can modify the autogenerated classes and your changes will be kept, even if you modify or rename the model in the extension builder. Always create backups in this mode or use a versioning system!

✓ (boolean)

Make a backup on each save [basic.backupExtension]

Should the extension builder make a complete backup of the extension, each time you save

✓ (boolean)

Backup dir [basic.backupDir]

Name of the backup dir (as subdir of path_SITE or as absolute path)

uploads/tx_extensionbuilder/backups (string)

Backups are stored in the directory `uploads/tx_extensionbuilder/backups` by default but this can be adjusted as required.

5.6.2. Modifying the Model

In contrast to the Kickstarter (TYPO3 extension that enables developers to create piBase extensions) the Extension Builder is capable of modifying the domain model at any time later – even if custom code has been written. This is an essential feature, because the DDD principle dictates that the code must be changeable if the model has been changed.

The central point of this feature is the YAML file `typo3conf/ext/simpleblog/Configuration/ExtensionBuilder/settings.yaml`.[15]

The content of this looks like the following by default:

```
#
# Extension Builder settings for extension simpleblog
# generated 2015-01-18T11:32:00Z
#
# See http://www.yaml.org/spec/1.2/spec.html
#

---

###########    Overwrite settings  ###########
#
# These settings only apply, if the roundtrip feature of the extension build
# is enabled in the extension manager
#
# Usage:
# nesting reflects the file structure
# a setting applies to a file or recursive to all files and subfolders
#
# merge:
#   means for classes: All properties ,methods and method bodies
#   of the existing class will be modified according to the new settings
#   but not overwritten
#
#   for locallang xlf files: Existing keys and labels are always
#   preserved (renaming a property or DomainObject will result in new keys a
#
#   for other files: You will find a Split token at the end of the file
#   After this token you can write whatever you want and it will be appended
#   everytime the code is generated
#
# keep:
#   files are never overwritten
#   These settings may break the functionality of the extension builder!
#   Handle with care!
#
#

############  extension settings  ##############

overwriteSettings:
  Classes:
    Controller: merge
    Domain:
      Model: merge
      Repository: merge

  Configuration:
    #TCA: merge
    #TypoScript: keep

  Resources:
    Private:
      #Language: merge
      #Templates: keep

  ext_icon.gif: keep

#  ext_localconf.php: merge

#  ext_tables.php: merge

#  ext_tables.sql: merge

## use static date attribute in xliff files ##
#staticDateInXliffFiles: 2015-01-18T11:32:00Z

## list of error codes for warnings that should be ignored ##
#ignoreWarnings:
  #503

######### settings for classBuilder #############################
#
# here you may define default parent classes for your classes
# these settings only apply for new generated classes
# you may also just change the parent class in the generated class file.
# It will be kept on next code generation, if the overwrite settings
# are configured to merge it
#
#################################################################

classBuilder:

  Controller:
    parentClass: \TYPO3\CMS\Extbase\Mvc\Controller\ActionController

  Model:
    AbstractEntity:
      parentClass: \TYPO3\CMS\Extbase\DomainObject\AbstractEntity

    AbstractValueObject:
      parentClass: \TYPO3\CMS\Extbase\DomainObject\AbstractValueObject

  Repository:
    parentClass: \TYPO3\CMS\Extbase\Persistence\Repository

  setDefaultValuesForClassProperties: true
```

`merge`

For classes, this means that all properties, methods and method bodies of the existing class will be modified according to the new settings but not overwritten. For language files, this means that existing keys and labels are always preserved (renaming a property or domain object will result in new keys and new labels). For other files, you will find a “split token” at the end of the file. Everything below that token remains unchanged: `## EXTENSION BUILDER DEFAULTS END TOKEN - Everything BEFORE this line is overwritten with the defaults of the extension builder`. The first part (letters in upper case) is sufficient. The end token is important especially when working with TCA files.

`keep`

If this keyword is used, the file will not be changed at all, even if the model has been changed.

`override`

This is the default setting, which means that the file will be re-written completely. As a consequence, for files not listed in any of these sections, `override` will be used.

5.6.3. Class Builder

By using the keyword `classBuilder:`, you may define the default parent classes for your classes. This is useful, if you want to implement your own classes at this point.

---

[13] http://cacoo.com

[14] https://forge.typo3.org/projects/extension-extension_builder/repository

[15] YAML (YAML Ain’t Markup Language) specification: http://www.yaml.org/spec/1.2/spec.html

Chapter 6. Preparation

Usually, extensions are embedded in TYPO3 CMS’ visual layout. In an “empty” TYPO3 instance, a frontend framework can be useful.

6.1. Frontend Frameworks

Out of a number of frameworks available today, two seem to have become universally accepted and established. Both frameworks also offer *Responsive Web Design*.

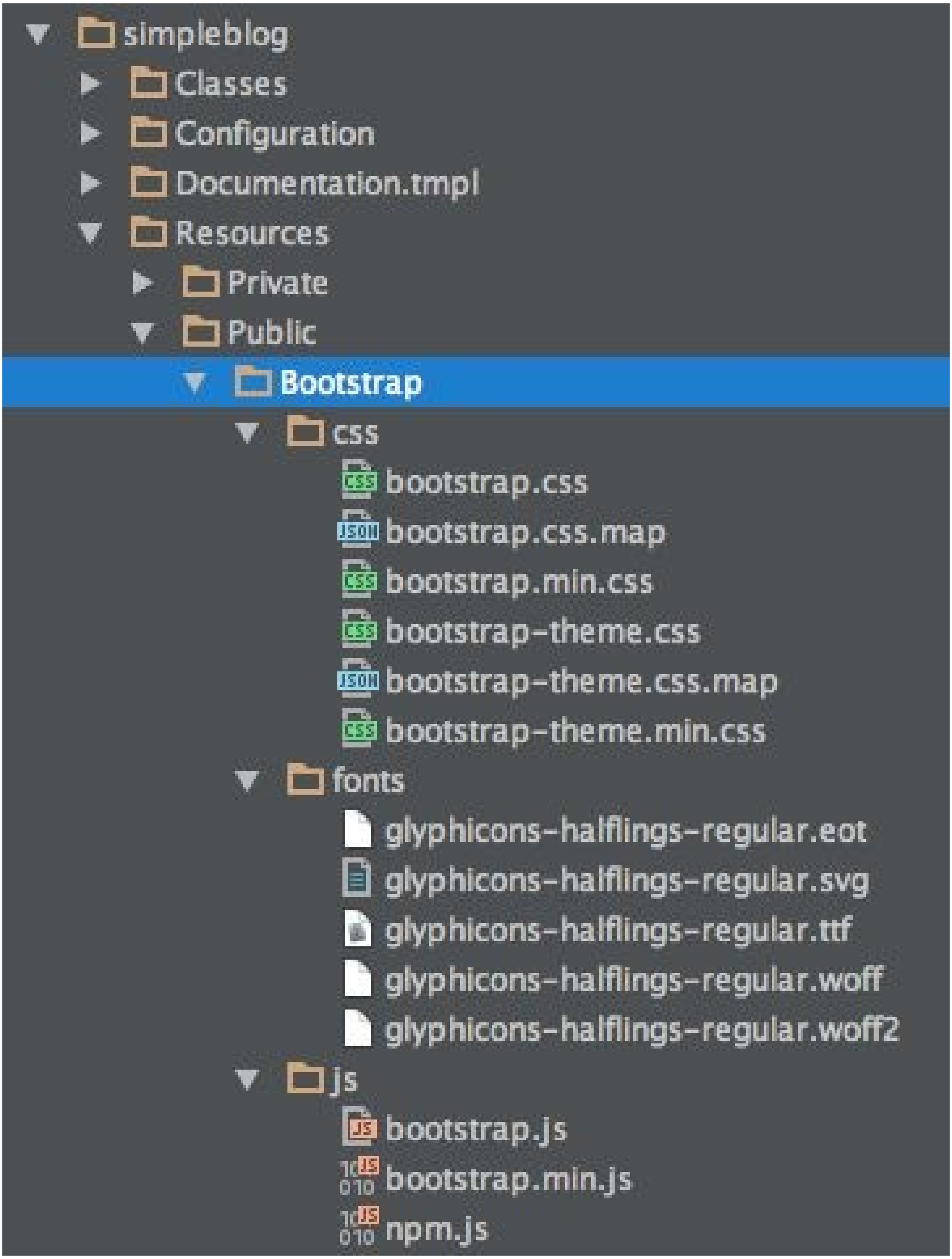
- Zurb Foundation (http://foundation.zurb.com)
- Twitter Bootstrap (http://getbootstrap.com)

We will use Twitter Bootstrap for our examples, which was released in March 2015 as version 3.3.4. This book is current for at least two years[16] because TYPO3 CMS 6.2 LTS will be supported by the TYPO3 Team until 2017 and the API will not change. This means, Bootstrap may experience further developments during this time but the basic implementation should remain the same.

First, download the Twitter Bootstrap package from the project website by following the link to the downloads. Then, extract the files into a directory.

In the next step, create a new folder `Boostrap` under `Resources/Public/` in the `simpleblog` extension and copy the directories `css`, `fonts` and `js` from the `dist` folder into the new Bootstrap folder.

Figure 6.1. Directory/file structure of the Twitter Bootstrap package



```
▼ simpleblog
  ▶ Classes
  ▶ Configuration
  ▶ Documentation.tmpl
  ▼ Resources
    ▶ Private
    ▼ Public
      ▼ Bootstrap
        ▼ css
            bootstrap.css
            bootstrap.css.map
            bootstrap.min.css
            bootstrap-theme.css
            bootstrap-theme.css.map
            bootstrap-theme.min.css
        ▼ fonts
            glyphicons-halflings-regular.eot
            glyphicons-halflings-regular.svg
            glyphicons-halflings-regular.ttf
            glyphicons-halflings-regular.woff
            glyphicons-halflings-regular.woff2
        ▼ js
            bootstrap.js
            bootstrap.min.js
            npm.js
```

Now open the file `Configuration/TypoScript/setup.txt` and add the following TypoScript code at the end of the file:

```
page {
   includeCSS {
      bootstrap = EXT:simpleblog/Resources/Public/Bootstrap/css/bootstrap.mi
      simpleblog = EXT:simpleblog/Resources/Public/Css/simpleblog.css
   }
   includeJSlibs {
      jquery = //code.jquery.com/jquery.js
      jquery.external = 1
      bootstrap = EXT:simpleblog/Resources/Public/Bootstrap/js/bootstrap.min
   }
}
```

This results in the inclusion of several files into the header of the website’s HTML: Bootstrap’s CSS file as well as a custom CSS file, then jQuery (using a CDN[17] as the source) and Bootstrap’s JavaScript library.

If you work locally and without an Internet connection, you can download jQuery, store it in the `js` folder and include it by amending the TypoScript code slightly:

```
      jquery = EXT:simpleblog/Resources/Public/Bootstrap/js/jquery.js
```

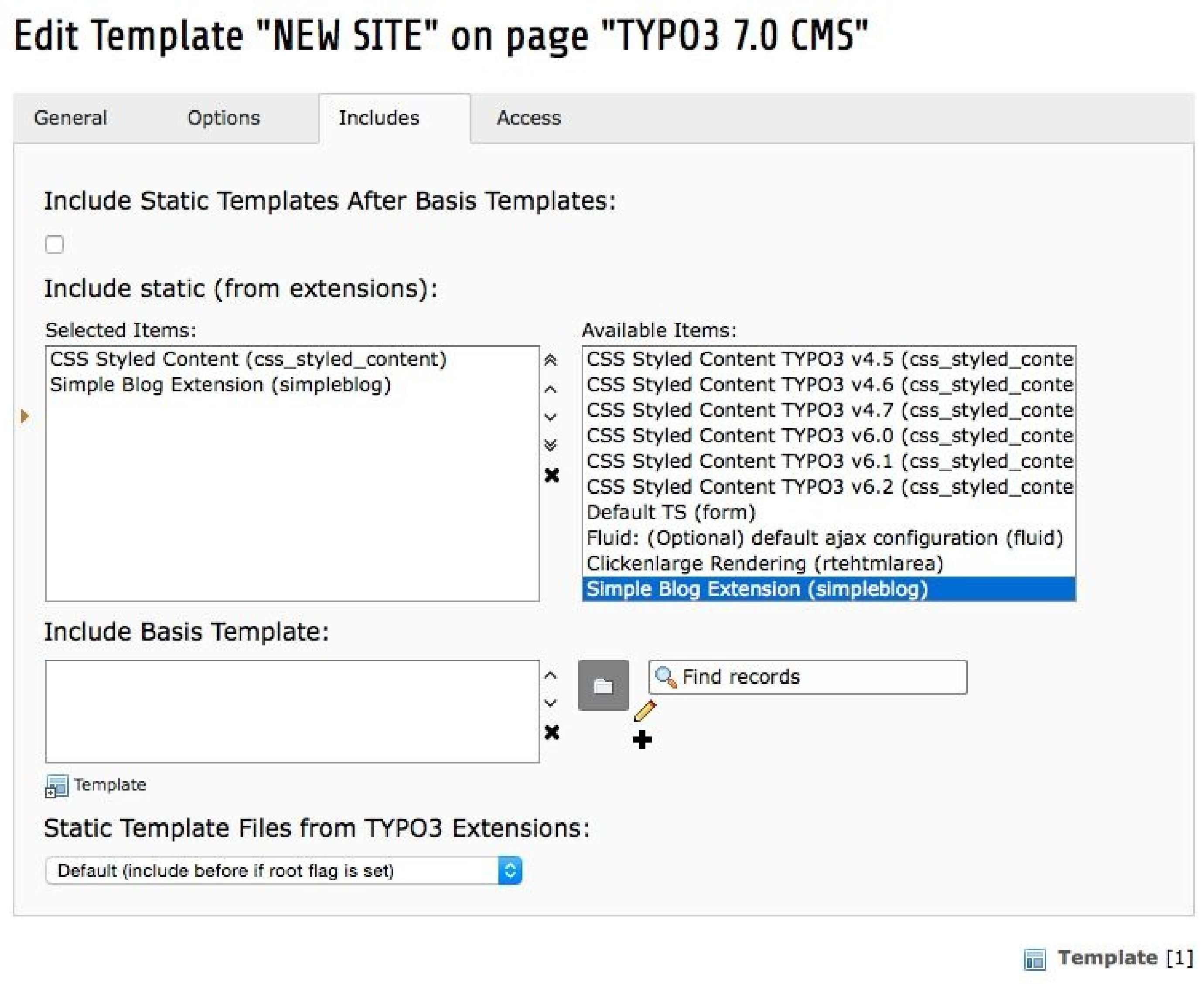
Bootstrap’s JavaScript folder is named `js` and we leave it as it stands. Although `Js` (UpperCamelCase) would be more consistent.

6.2. Load Static TypoScript

In order to let TYPO3 load the TypoScript, we have to include it first:

Access module *Template* in TYPO3's backend and go to the root template of the website. Select *Info/Modify* from the dropdown box at the top (function menu) and follow the link *Edit the whole template record*. Now switch to tab *Includes* and choose *Simpleblog Extension (simpleblog)* from *Available Items* (right) under the *Include static (from extensions)* section. The item appears in the left field. Finally, save the template record.

Figure 6.2. Load static template of extension `simpleblog`



Edit Template "NEW SITE" on page "TYPO3 7.0 CMS"

Include Static Templates After Basis Templates:

Include static (from extensions):

Selected Items:

- CSS Styled Content (css_styled_content)
- Simple Blog Extension (simpleblog)

Available Items:

- CSS Styled Content TYPO3 v4.5 (css_styled_conte
- CSS Styled Content TYPO3 v4.6 (css_styled_conte
- CSS Styled Content TYPO3 v4.7 (css_styled_conte
- CSS Styled Content TYPO3 v6.0 (css_styled_conte
- CSS Styled Content TYPO3 v6.1 (css_styled_conte
- CSS Styled Content TYPO3 v6.2 (css_styled_conte
- Default TS (form)
- Fluid: (Optional) default ajax configuration (fluid)
- Clickenlarge Rendering (rtehtmlarea)
- Simple Blog Extension (simpleblog)

Include Basis Template:

Static Template Files from TYPO3 Extensions:

Default (include before if root flag is set)

6.3. Load CSS File

Create a new directory `Resources/Public/Css/` and add a file `simpleblog.css` with the content below:

```
body {
        padding: 10px;
}
```

6.4. IDE Settings

Extbase & Fluid software development requires a powerful IDE (“Integrated Development Environment”). A wide range of available IDEs exists – for example the following projects have no charge:

- NetBeans IDE[18]
- PDT/Eclipse[19]

Commercial, non-free IDEs are for example:

- Komodo IDE[20]
- PhpED[21]
- PhpStorm[22]
- Zend Studio[23]

PhpStorm is highly recommended due to its wide acceptance in the community, the affordable price (99 EUR for individual developers and 199 EUR for companies and organisations – as of March 2015), the immense functionality, the intuitive user interface and the great performance and many agencies work with PhpStorm too.

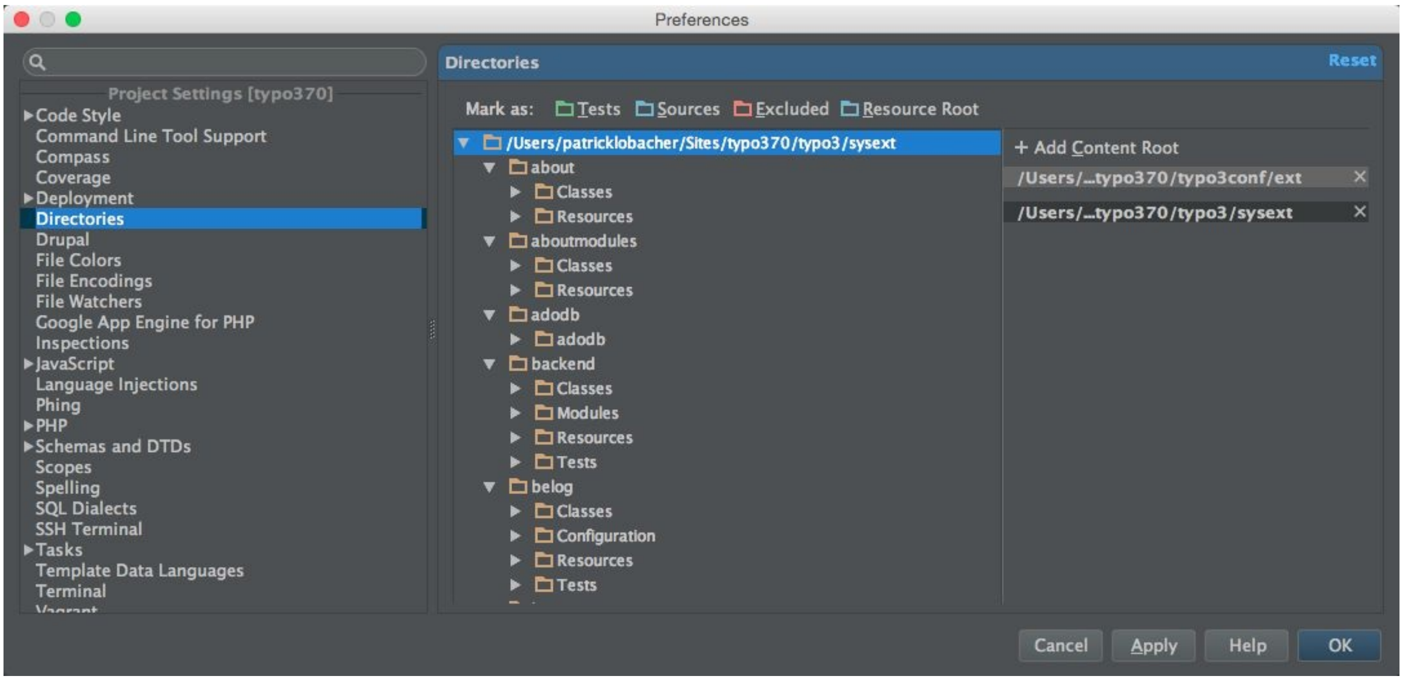
For those not wanting to invest money in an IDE, they should consider NetBeans IDE.

Before you start developing your software, there needs to be some configuration of the IDE.

6.4.1. Add Core Files to the Include Path

Open the preferences of PhpStorm and locate the *Project Settings (ext)* in section *Directories*. Add folder `typo3/sysext/` to the list of directories.

Figure 6.3. Add TYPO3 core files to the include path



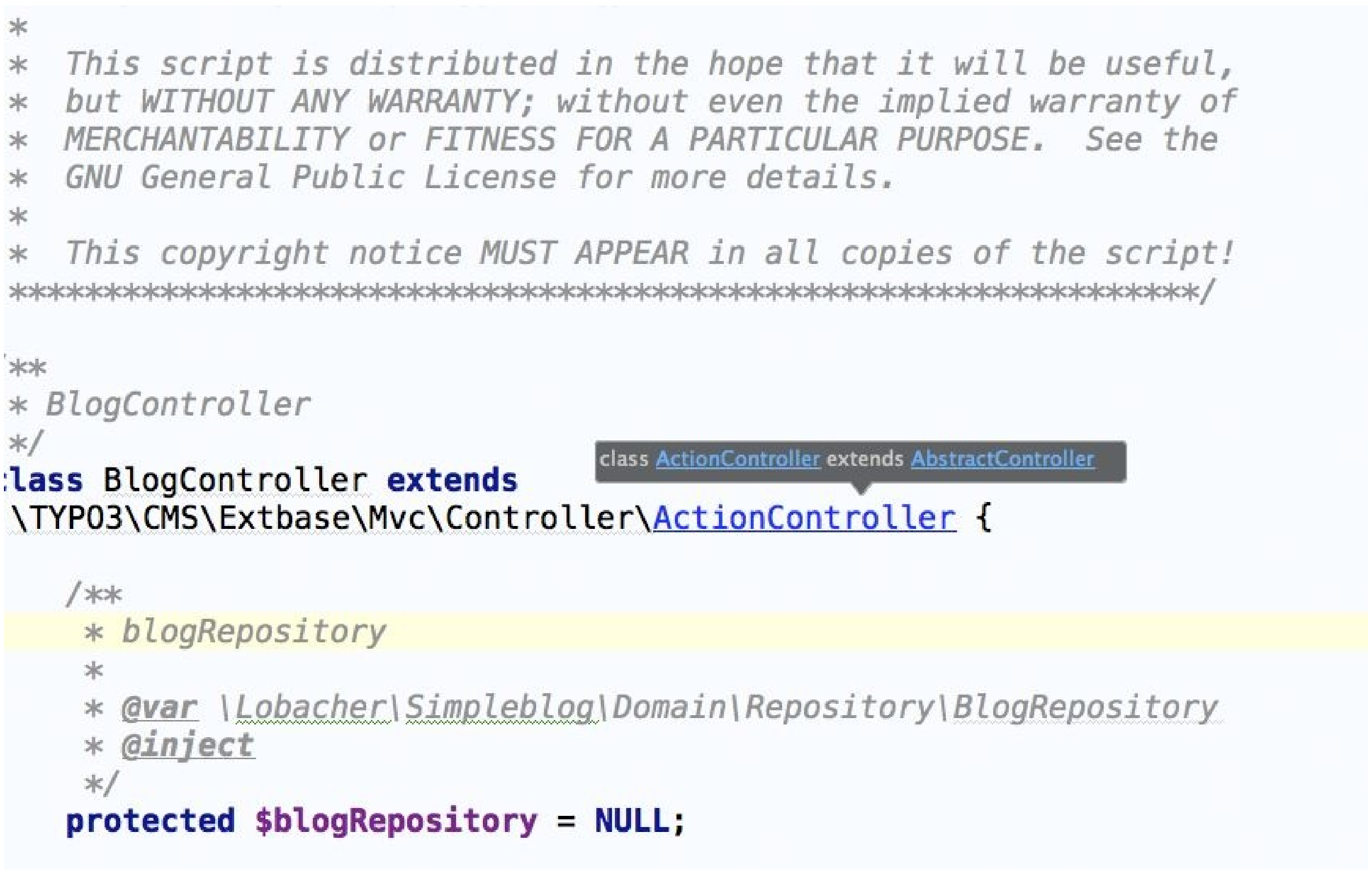
```
▼ /Users/patricklobacher/Sites/typo370/typo3/sysext
  ▼ about
    ▶ Classes
    ▶ Resources
  ▼ aboutmodules
    ▶ Classes
    ▶ Resources
  ▼ adodb
    ▶ adodb
  ▼ backend
    ▶ Classes
    ▶ Modules
    ▶ Resources
    ▶ Tests
  ▼ belog
    ▶ Classes
    ▶ Configuration
    ▶ Resources
    ▶ Tests
```

```
/Users/...typo370/typo3conf/ext
/Users/...typo370/typo3/sysext
```

Why is it important to add this directory?

When you are working on the source code of the BlogController file for example, you can press the key `cmd` (Mac) or `Ctrl` and move your mouse pointer over a class name (e.g. `ActionController`). PhpStorm immediately shows where the source of the definition is (`class ActionController extends AbstractController`). A click on this opens the original file and as a result of this feature, you can navigate through the code quickly.

Figure 6.4. Show source code class definition



```
*
*  This script is distributed in the hope that it will be useful,
*  but WITHOUT ANY WARRANTY; without even the implied warranty of
*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*  GNU General Public License for more details.
*
*  This copyright notice MUST APPEAR in all copies of the script!
****************************************************************/

**
* BlogController
*/
lass BlogController extends
\TYPO3\CMS\Extbase\Mvc\Controller\ActionController {

   /**
    * blogRepository
    *
    * @var \Lobacher\Simpleblog\Domain\Repository\BlogRepository
    * @inject
    */
   protected $blogRepository = NULL;
```

6.4.2. Include Fluid’s Schema File

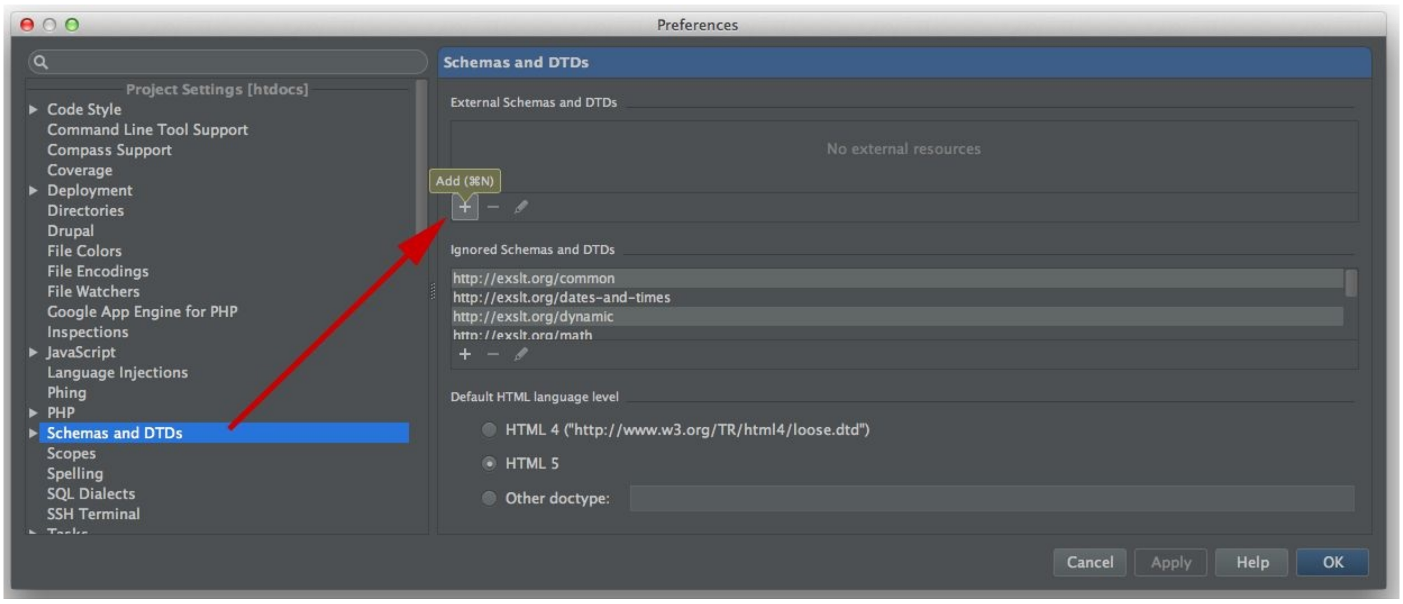
While the IDE is able to resolve class and interface names automatically, this does not work for Fluid. Fluid is a propriety technique and we have to teach the IDE the appropriate syntax highlighting and auto-completion functionality first.

Luckily, a XSD schema file is available, which we can simply include in PhpStorm (or any other IDE software that supports this).

First, download the XSD file for the TYPO3/Extbase version you are using.[24]

Now open *Schemas and DTDs* in PhpStorm’s *Preferences* and click the plus symbol under *External Schemas and DTDs*.

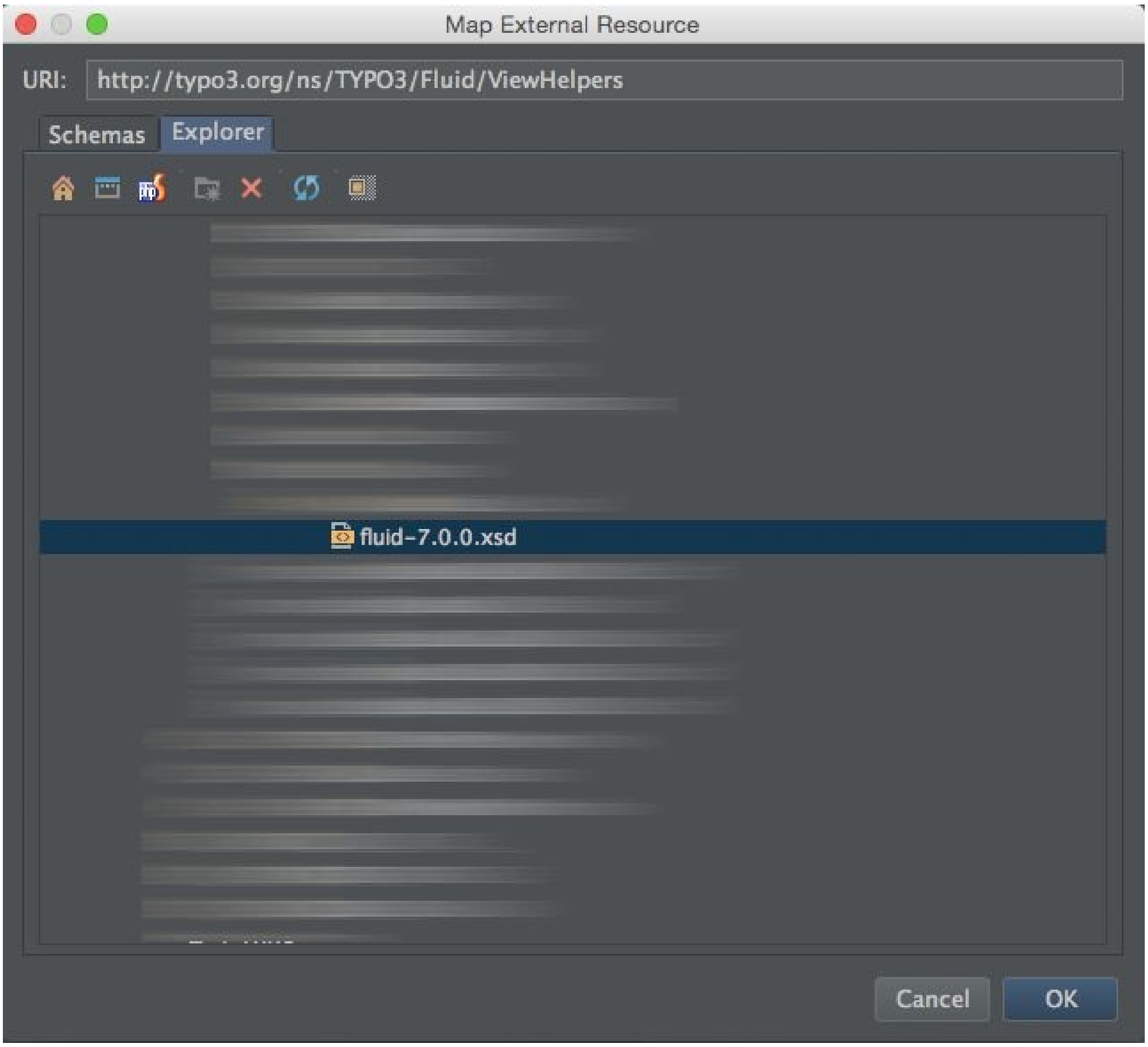
Figure 6.5. Add a schema file



Default HTML language level:
HTML 4 ("http://www.w3.org/TR/html4/loose.dtd") [✓ HTML 5] Other doctype:

After that, open the tab *Explorer* and locate the file on your local computer that you downloaded before. In addition, enter the following value in the URL field: `http://typo3.org/ns/TYPO3/Fluid/ViewHelpers`.

Figure 6.6. Enter URI and select the schema file



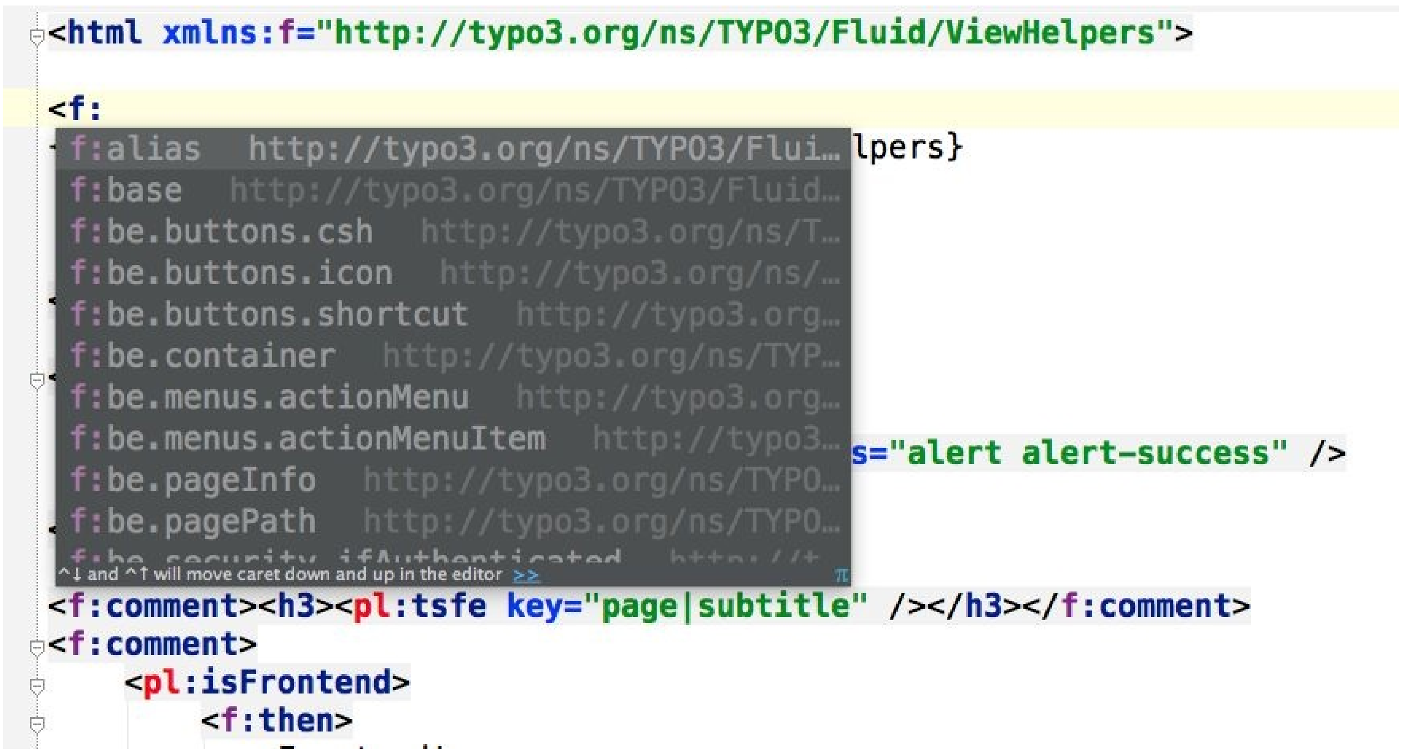
URI: http://typo3.org/ns/TYPO3/Fluid/ViewHelpers

fluid-7.0.0.xsd

Finally, save your configuration changes by clicking on *OK*. That is all it takes for the schema file to be used. Add the following lines to a HTML file and PhpStorm responds with an auto-suggestion:

```
<html xmlns:f="http://typo3.org/ns/TYPO3/Fluid/ViewHelpers">
...
</html>
```

Figure 6.7. Auto-suggestion in the IDE



```
<html xmlns:f="http://typo3.org/ns/TYPO3/Fluid/ViewHelpers">

<f:
 f:alias   http://typo3.org/ns/TYPO3/Flui…  lpers}
 f:base   http://typo3.org/ns/TYPO3/Fluid…
 f:be.buttons.csh   http://typo3.org/ns/T…
 f:be.buttons.icon   http://typo3.org/ns/…
 f:be.buttons.shortcut   http://typo3.org…
 f:be.container   http://typo3.org/ns/TYP…
 f:be.menus.actionMenu   http://typo3.org…
 f:be.menus.actionMenuItem   http://typo3…  s="alert alert-success" />
 f:be.pageInfo   http://typo3.org/ns/TYPO…
 f:be.pagePath   http://typo3.org/ns/TYPO…
 ^↓ and ^↑ will move caret down and up in the editor >>
<f:comment><h3><pl:tsfe key="page|subtitle" /></h3></f:comment>
<f:comment>
    <pl:isFrontend>
        <f:then>
```

To prevent the display of the `<html>`-tag in the content, you should place this outside of `<section>`:

```
<html xmlns:f="http://typo3.org/ns/fluid/ViewHelpers">

<f:layout name="defaultLayout" />

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

</html>
```

6.4.3. Tips About Resolving Class Names

By including classes the “normal way”, PhpStorms resolves their names without problems. However if you try using the ObjectManager for example, this will not work.

```
$validatorResolver = $this->objectManager->get('Tx_Extbase_Validation_Valida
```

The methods and properties of the `ValidatorResolver` are completely invisible to PhpStorm. This can be solved by adding the `@var`-annotation, which builds the reference to the class. After that, PhpStorm is able to resolve the data correctly:

```
/** @var \TYPO3\CMS\Extbase\Validation\ValidatorResolver $validatorResolver
$validatorResolver = $this->objectManager->get('Tx_Extbase_Validation_Valida
```

6.4.4. TYPO3 Extension phpstorm

PhpStorm supports a meta data file `.php-storm.meta.php` since version 6.0.1, which supports the IDE in resolving factory methods.

Ingo Renner developed the TYPO3 extension *PhpStorm Meta Data* (extension key: `phpstorm`), which is available in the TER[25] and at GitHub.[26] This extension generates exactly this meta data file. The following factory methods are supported:

- `TYPO3\CMS\Core\Utility\GeneralUtility::makeInstance`
- `TYPO3\CMS\Extbase\Object\ObjectManager::create`
- `TYPO3\CMS\Extbase\Object\ObjectManager::get`

In order to activate this feature, follow the steps below:

- Create a new backend user named `_cli_phpstorm` (use a password of your choice)
- Execute the following command on the command line:

  ```
  typo3/cli_dispatch.phpsh phpstorm_metadata
  ```

A new file with the name `.phpstorm.meta.php` has been created and shows the following content (or similar):

```
<?php
namespace PHPSTORM_META {
   /** @noinspection PhpUnusedLocalVariableInspection */
   /** @noinspection PhpIllegalArrayKeyTypeInspection */
   $STATIC_METHOD_TYPES = [
      \t3lib_div::makeInstance('') => [
         'Buch\\Extbase\\Code\\Car' instanceof \Buch\Extbase\Code\Car,
         'localPageTree' instanceof \localPageTree,
         'idna_convert' instanceof \idna_convert,
         'HTTP_Request2_Adapter_Curl' instanceof \HTTP_Request2_Adapter_Curl
```

The following parameters can be passed to the script:

`--disableClassAliases`
: Namespaces have been introduced with TYPO3 version 6.0. To ensure backward compatibility, `Class Alias Maps` ensure that old names can still be used and are mapped to the appropriate new names. This can be disabled by using this parameter.

`-s` / `--silent`
: Using this parameter, only errors and important warnings are shown.

`-ss`
: No output is shown at all.

But now we are really ready to start exploring Extbase and Fluid!

[16] Likely much longer, because we also took TYPO3 CMS 7.x into account, but the LTS version of this release is scheduled after this book has been published.

[17] CDN: Content Delivery Network

[18] https://netbeans.org/

[19] http://projects.eclipse.org/projects/tools.pdt

[20] http://www.activestate.com/komodo-ide/downloads

[21] http://www.nusphere.com/download.php.ide.htm#phped

[22] http://www.jetbrains.com/phpstorm/

[23] http://www.zend.com/de/products/studio/

[24] http://www.extbase-book.org/resources.html

[25] http://typo3.org/extensions/repository/view/phpstorm

[26] https://github.com/irnnr/typo3-ext-phpstorm

Chapter 7. The CRUD Process

The “CRUD process” is important when it comes to an object life cycle, this process controls the four steps of an object which are:

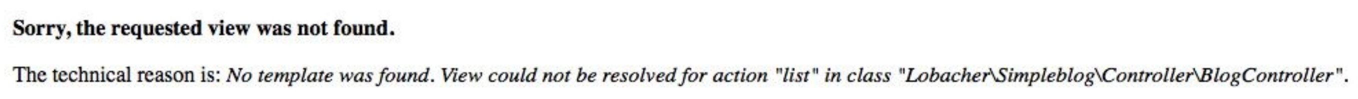
- Create
- Read
- Update
- Delete

Below we take a closer look at these steps by using the example of a Blog object:

7.1. Creating an Object

If we access the extension in the frontend as it stands now, a meaningful error message occurs (Figure 7.1).

Figure 7.1. Error message: no template found



Sorry, the requested view was not found.

The technical reason is: *No template was found. View could not be resolved for action "list" in class "Lobacher\Simpleblog\Controller\BlogController".*

The appropriate template is missing, we will create this in the next step.

But let's have a look at the generated code first.

7.1.1. Implementing listAction

Open file `typo3conf/ext/simpleblog/Classes/Controller/BlogController.php`:

```
<?php
namespace Lobacher\Simpleblog\Controller;

class BlogController extends \TYPO3\CMS\Extbase\Mvc\Controller\ActionControl

        /**
         * blogRepository
         *
         * @var \Lobacher\Simpleblog\Domain\Repository\BlogRepository
         * @inject
         */
        protected $blogRepository;

        /**
         * action
         *
         * @return void
         */
        public function listAction() {

        }

}
?>
```

Here we can see, that the Blog-Repository is loaded by using Dependency Injection – we will return to this later.

Two things are important here, first, the fact that the controller class has been deviated from a generic controller `\TYPO3\CMS\Extbase\Mvc\Controller\ActionController` (it is worth reviewing this class to understand what has been prepared in the controller, which is not visible directly). Second, the method `listAction()`.

The annotation `@return` specifies the data type of the return value. As we do not want to return anything, we choose the type `void`.

The error message above shows exactly, what we have to do to fix the issue: the appropriate template is missing, so we will create this in the next step.

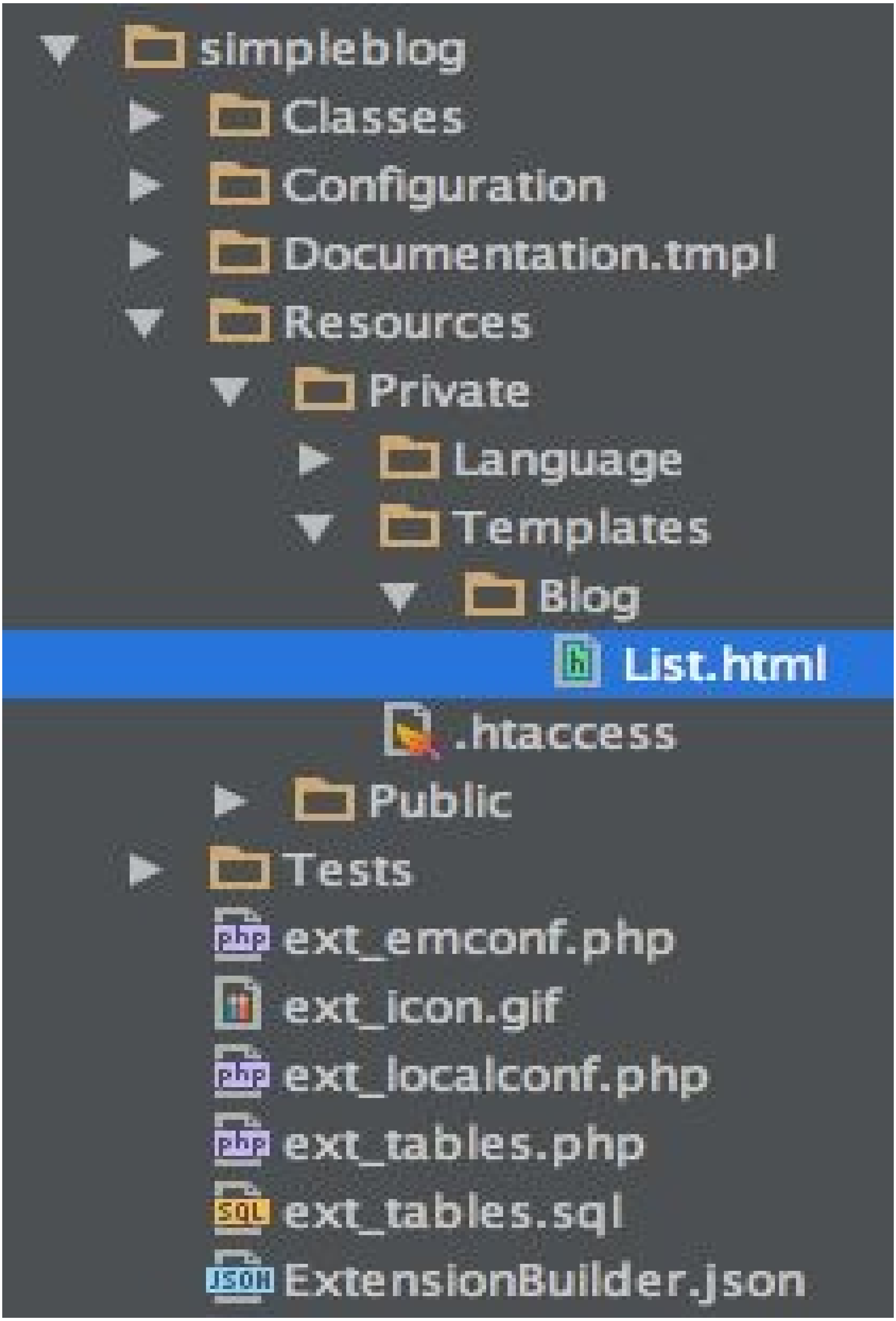
7.1.2. Creating the Template of listAction

Create a new directory `Templates` under `typo3conf/ext/simpleblog/Resources/Private/` and inside this, a directory called `Blog`.

One of Extbase's strict conventions is that sub-directories of `Templates` are always named as the domain object. Another convention says that files inside this directory are named as the actions in UpperCamelCase and their file extension is `.html` (by default).

Therefore, we create a new file `List.html` in directory `Blog`.

Figure 7.2. Directory structure after creating file `List.html`



```
▼ simpleblog
    ▶ Classes
    ▶ Configuration
    ▶ Documentation.tmpl
    ▼ Resources
        ▼ Private
            ▶ Language
            ▼ Templates
                ▼ Blog
                    List.html
              .htaccess
        ▶ Public
    ▶ Tests
      ext_emconf.php
      ext_icon.gif
      ext_localconf.php
      ext_tables.php
      ext_tables.sql
      ExtensionBuilder.json
```

We add the following simple line as the content of file `List.html`:

```
<h1>Blog-List</h1>
```

If we access the frontend again, we can see by the output of the template that the request processed successfully.

7.1.3. Side Note: Template Rendering

Template rendering happens at the end of the action automatically by executing the following function:

```
return $this->view->render();
```

Only if you explicitly `return FALSE`, the View is not being called and as a consequence, no output is returned. If you return a string, the string is shown as the output. In order to return an object, method `__toString()` can be called in the object and the string is returned.

Render a view without output

In order to render a view (e.g. to render an email template, which can be used as the mail body by the `mail()` function), you can call the render method:

```
$mailBody = $this->view->render()
```

At the end of the action you could then either `return FALSE` or redirect to another action.

7.1.4. Create Static Blogs

In the first step, we use a loop to create three Blogs in the `listAction` and we set the title to “This is the 1. Blog” (etc.) by using the getter. Then, we assign the created Blog to the array variable `blogs` and the variable to the view by using method `assign()`.

```
public function listAction() {
        $blogs = array();
        for ($i=1; $i<=3; $i++) {
                $blog = $this->objectManager->get('\\Lobacher\\Simpleblog\\D
                $blog->setTitle('This is Blog number ' . $i . '!');
                $blogs[] = $blog;
        }
        $this->view->assign('blogs', $blogs);
}
```

The view receives the array and outputs it. To do this, open file `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/List.html` and add the following code:

```
<h1>Blog List</h1>

<ul class="list-group">
    <f:for each="{blogs}" as="blog">
        <li class="list-group-item">{blog.title}</li>
    </f:for>
</ul>
```

We wrapped the unordered list with `<ul>...</ul>` for the output. Inside, we use a Fluid-ViewHelper which implements a loop. `each="{blogs}"` defines which elements the loop iterates. By stating `{blogs}` we indicate that we want to use the array which we have assigned in the controller. The name used by method `assign()` in the controller is the same as the name used in curly brackets in the view.

Parameter `as="blog"` specifies that a single element inside the loop is named `blog`. Accessing this can be done by using `{blog}` (but only inside the loop). The `<li>...</li>` code makes use of that: in order to output the title of the Blog, we are using `{blog.title}`, which calls the getter to retrieve the property `title` of the object `blog`.

As a result, the three Blogs are shown as an unordered list.

7.1.5. Persisting the Blogs

If we access the extension in the frontend, we see the list of Blogs. This list remains the same every time we access it because the same Blogs are created over and over again. Secondly, we have not made the Blogs persistive yet – this will be our next step.

According to DDD we have to use a repository for that. A closer look at the Blog controller reveals that a repository has already been prepared:

```
/**
 * blogRepository
 *
 * @var \Lobacher\Simpleblog\Domain\Repository\BlogRepository
 * @inject
 */
protected $blogRepository;
```

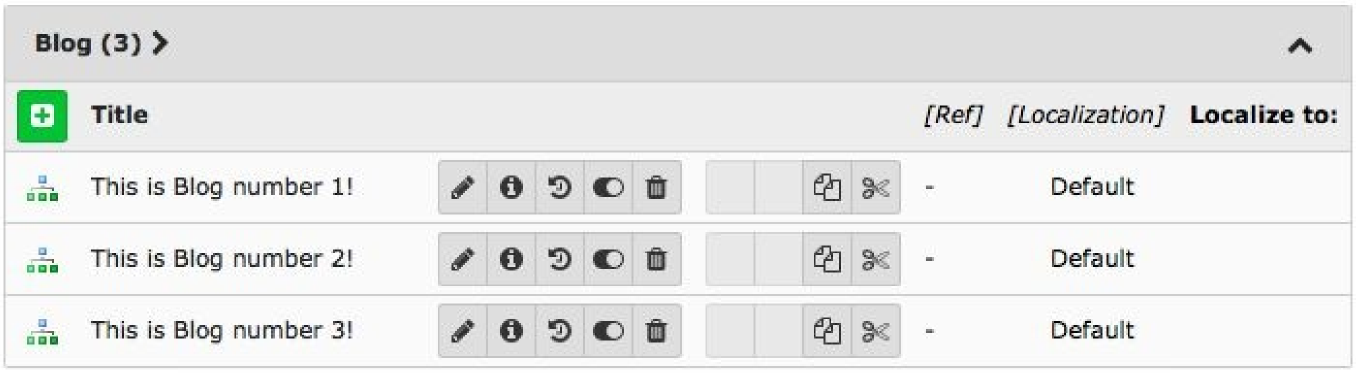
These few lines are sufficient to fetch the class `\Lobacher\Simpleblog\Domain\Repository\BlogRepository` (or a different implementation, which could be configured by using TypoScript) via Dependency Injection (the combination of the annotations `@inject` and `@var` take care of that) and assign it to the property `$blogRepository`.

A repository has a number of methods – the most interesting is `add()` at this point in time because we want to add something to the repository. Let's update our code so that every Blog we create is added to the repository and at the end of the action we access the repository to retrieve the stored Blogs and pass them to the view. The method `findAll()` does exactly that:

```
public function listAction() {
        $blogs = array();
        for ($i=1; $i<=3; $i++) {
                $blog = $this->objectManager->get('\\Lobacher\\Simpl
                $blog->setTitle('This is Blog number ' . $i . '!');
                $this->blogRepository->add($blog);
        }
        $this->view->assign('blogs',$this->blogRepository->findAll()
}
```

If we access the extension in the frontend again (after clearing the cache to play it safe), nothing is shown but if we look at the backend, something interesting has happened: go to the page UID `0` (the page with the TYPO3 icon pre-pended) and open the *List* module. Here you will see that all Blogs are stored successfully.

Figure 7.3. Blogs listed in the backend on page UID `0`



Blog (3)

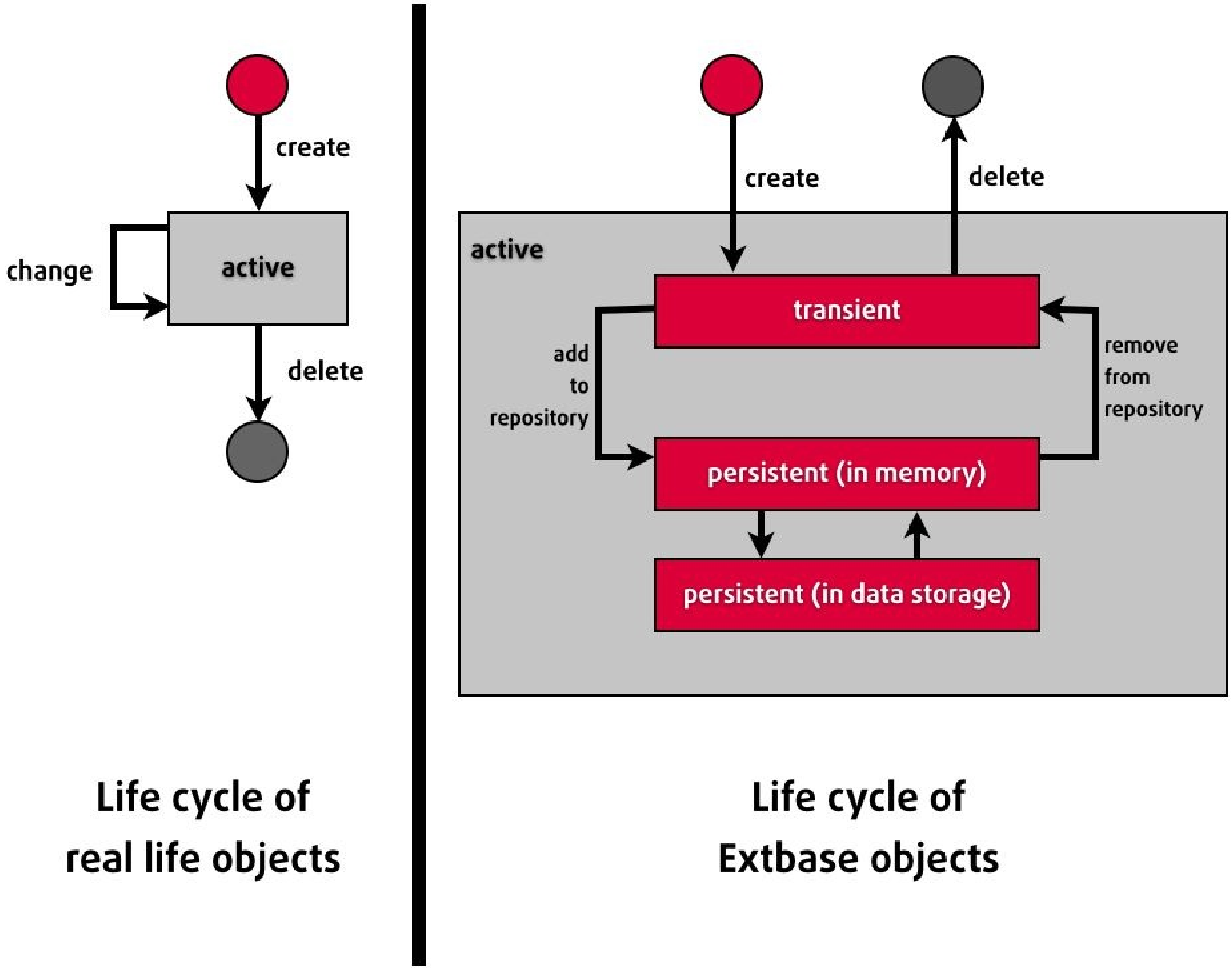
Title	[Ref]	[Localization]	Localize to:
This is Blog number 1!	-	Default	
This is Blog number 2!	-	Default	
This is Blog number 3!	-	Default	

Accessing the extension a second time shows three Blogs in the frontend compared to six list entries in the backend. This mystery needs to be unravelled!

7.1.6. Side Note: Persistence

In “normal life” the life cycle of an object is quite simple, it’s created, it’s active and then disappears again. The same applies to humans as well as refrigerators.

Figure 7.4. Life cycle of objects



We will take a look at this cycle in Extbase:

As before, firstly the object is created and in an *active* state (this is also called *transitive*).

When an object is added to a repository, it is saved to the memory first. Only transitive objects (objects that exist in the memory) can be deleted. If an object has been persisted already, it must be transferred from the database to a transitive state first. Only then can it be deleted from the memory and added to a list of objects for deletion. Later, when the objects are being made persistent, this same list is processed and the objects removed from the database.

In general, Extbase persists at the end of an action, which means the following steps happen as a kind of “cashing-up”:

- If an object should be created (for example because the `add()` method has been applied to the repository), this is noted in a list first.
- If one of these objects changed, this is also noted in this list.
- If an object has been removed from the repository, Extbase checks first, if it has been earmarked to be created or altered – in this case, these steps became irrelevant.

This technique is highly efficient because only those objects that are created, updated or deleted are really used. As a side effect, access to the data storage is minimised.

However a drawback of this approach is that you are unable to tell which operations have been executed at the end. Things like `mysql_insert_id` do not exist any more. Also this system only performs at its best, if it is used rarely. As a result of this, persistence is only used at the end of an action in Extbase.

In our example Extbase executes the steps described below, when adding an object to the repository or removing an object from it:

- Read access to the repository (e.g. `findAll()`) always happens at this position in the code, where the methods appear.
- Write access to the repository (e.g. `add()`) always happens at the end of the action.

Hence the reason, there are always three objects more in the database than can be visible at the frontend. To avoid this, we have to manually interfere and force to persist.

Implicit Persistence

The implicit persistence at the end of every action has been removed in TYPO3 CMS 6.2 LTS. Persistence only happens after an appropriate call to the repository, e.g. `add()` or `update()`.

7.1.7. The Persistence Manager: Manual Persistence

Extbase features a *Persistence Manager* in file `typo3/sysext/extbase/Classes/Persistence/Generic/PersistenceManager.php`. Please open this file in your editor and investigate it.

Public API

All methods of the *public API* can be used in your Extbase code. To identify these methods, look out for the annotation `@api`. They exist in classes in there, before methods. If you come across a method which does not have an `@api` annotation, you should not use this in your code because the API may change without warning at any time.

```
<?php
namespace TYPO3\CMS\Extbase\Persistence\Generic;
...
/**
 * The Extbase Persistence Manager
 *
 * @api
 */
class PersistenceManager implements \TYPO3\CMS\Extbase\Persistence\Persisten
   ...
        /**
         * Commits new objects and changes to objects in the current persist
         * session into the backend
         *
         * @return void
         * @api
         */
        public function persistAll() {
        ...
```

Here you can see the method `persistAll()`, which takes care of the manual persistiveness when called. In order to trigger this method, we have to do two things:

1. Load the class via Dependency Injection
2. Call the method prior end of action

Let’s go back to the BlogController:

```
        ...
        protected $blogRepository;

        /**
         * Persistence Manager
         *
         * @var \TYPO3\CMS\Extbase\Persistence\Generic\PersistenceManager
         * @inject
         */
        protected $persistenceManager;

        /**
         * list action
         *
         * @return void
         */
        public function listAction() {
                $blogs = array();
                for ($i=1; $i<=3; $i++) {
                        $blog = $this->objectManager->get('\\Lobacher\\Simpl
                        $blog->setTitle('This is Blog number ' . $i . '!');
                        $this->blogRepository->add($blog);
                }
                $this->persistenceManager->persistAll();
                $this->view->assign('blogs',$this->blogRepository->findAll()
        }
        ...
```

Now the number of Blogs shown in the frontend matches the number of records in the repository.

7.1.8. Create Your Own Action

At the moment, the action `listAction` carries out two tasks: it creates the objects and generates the list of Blogs. Strictly speaking, this is not in accordance with Extbase's concept of a *Slim Controller*: multiple tasks should be implemented in different actions. This presumes the following:

1. We have to register an additional action in file `ext_localconf.php` and choose to use `add` as its name.
2. In the controller class, a new method `addAction` is required, where we implement the code to generate Blogs.
3. The remaining code in the original method `listAction()` contains the logic of querying the repository only and passing the data over to the view.
4. Finally, we slightly extend the code in `addAction()` and add a redirect to `listAction()`, so that we end up at the list view after creating Blogs.

Let's start with file `ext_localconf.php`:

```
\TYPO3\CMS\Extbase\Utility\ExtensionUtility::configurePlugin(
        'Lobacher.' . $_EXTKEY,
        'Bloglisting',
        array(
                'Blog' => 'list, add',

        ),
        // non-cacheable actions
        array(
                'Blog' => 'list, add',

        )
);
```

Followed by the new `addAction()` in BlogController:

```
   /**
    * add action
    *
    * @return void
    */
   public function addAction() {
      for ($i=1; $i<=3; $i++) {
         $blog = $this->objectManager->get('\\Lobacher\\Simpleblog\\Domain\\
         $blog->setTitle('This is Blog number ' . $i . '!');
         $this->blogRepository->add($blog);
      }
      $this->redirect('list');
   }
```

The manual persistence via `$this->persistenceManager->persistAll();` can be left out because the data is made persistent automatically at the end of the action.

Method `$this->redirect();` triggers a new request to action `list`. This is a HTTP request, which means, the page is completely reloaded and therefore the URL changes as well.

Redirect

The method `$this->redirect()` features a lot of parameters, which will be used frequently in this book:

```
/**
 * Redirects the request to another action and / or controller.
 *
 * Redirect will be sent to the client which then performs another requ
 *
 * NOTE: This method only supports web requests and will thrown an exce
 * if used with other request types.
 *
 * @param string $actionName Name of the action to forward to
 * @param string $controllerName Unqualified object name of the control
 * @param string $extensionName Name of the extension containing the co
 * @param array $arguments Arguments to pass to the target action
 * @param integer $pageUid Target page uid. If NULL, the current page u
 * @param integer $delay (optional) The delay in seconds. Default is no
 * @param integer $statusCode (optional) The HTTP status code for the r
 * @return void
 * @api
 */
protected function redirect($actionName, $controllerName = NULL, $exten
```

The first parameter is the action, the second the controller, the third the extension and so forth. If parameters are not stated in the method call, their default values are used, e.g. the current controller name.

Finally, we purge the code in `listAction()` of the BlogController so that it only retrieves/displays Blogs but does not create any:

```
/**
 * list action
 *
 * @return void
 */
public function listAction() {
   $this->view->assign('blogs',$this->blogRepository->findAll());
}
```

In order to enable users to create new Blogs, we have to add a link to the view `List.html`, ideally at the very end of the file:

```
...
<f:link.action action="add" class="btn btn-primary">Add Blogs</f:link.action
```

This line generates an `<a href="...">` tag, which already points to the right action. Checking the URL of this tag more closely reveals how it works:

```
http://www.example.com/index.php?id=1&
   tx_simpleblog_bloglisting[action]=add&
   tx_simpleblog_bloglisting[controller]=Blog&
   cHash=3d6a8fa3884f23198e0cb9a1604abccb
```

As above, two parameters show a name space, which contains the extension key as well as the plugin name. Both parameters include information about the action and the controller, which tell Extbase to access the action `addAction()` in BlogController.

In addition, a parameters `cHash` (also known as “Cache Hash”) exists. Its value is an MD5 hash, which has been calculated based on the parameters and TYPO3’s encryption key. Only if the cHash and the parameters match, can the page be shown and written to the cache. This only applies to parameters generated by extensions.[27]

7.1.9. Form to Create an Object

Certainly the creation of objects via `new()` and `$this->objectManager->get('...');` is reasonable and useful. However a form which allows us to create Blogs would be more efficient.

In this case, we have to reconsider something from a conceptional perspective: in theory it would be possible to put the form as well as the creation of Blogs in one action but this would have drawbacks. Firstly, we would have too much logic in the controller, this only checks the status (shows empty form, accepts form data, validates form, etc.). Secondly, as pointed out before, Extbase follows the *Slim Controller* principle.

Therefore, we will implement the creation of an object via a form as a two-step action.

First, let’s add the `addForm` action prior to the `add` action in file `ext_localconf.php`:

```
...
   array(
      'Blog' => 'list, addForm, add',
   ),
   // non-cacheable actions
   array(
      'Blog' => 'list, addForm, add',
   )
...
```

Now a method `addFormAction()` is required in the controller:

```
   /**
    * addForm action - displays a form for adding a blog
    *
    * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
    */
   public function addFormAction(\Lobacher\Simpleblog\Domain\Model\Blog $blo
      $this->view->assign('blog',$blog);
   }
```

This will result in the below factors:

- The method has an input parameter `$blog`.
- This parameter must be of type `\Lobacher\Simpleblog\Domain\Model\Blog` because this has been specified as a type hint.
- An annotation `@param` indicates that `$blog` is of type `\Lobacher\Simpleblog\Domain\Model\Blog`.
- The object is passed to the view in this method.

This combination of the type hint, the annotation and the existence of a key with the same name in the request ensures that an object of the type of the domain object Blog can be reconstituted and assigned to variable `$blog`.

With the first request, nothing is restored but a new object is being built. For this purpose, a new instance of the domain class is generated and the value `NULL` assigned to the object. This is required as we do not have an object at the first call request. At the same time, we want to pass the object to the view and this is why the initialisation must happen.

Let's attend to the view.

Side Note: Fluid

Before we build up the form, some general notes about Fluid: Fluid is a template engine, which replaces the traditional marker/sub-part approach. This has been used in the *piBase* extension development, which came along as the following:

```
// Determine the template
$this->templateCode = $this->cObj->fileResource($conf['templateFile']);

// Read sub-part
$template['total'] = $this->cObj->getSubpart($this->templateCode,'###TEMPLAT

// Fill markers
$markerArray['###MARKER1###'] = 'content for marker 1';
$markerArray['###MARKER2###'] = 'content for marker 2';

// Set markers in template
$content = $this->cObj->substituteMarkerArrayCached($template['total'],
$markerArray);
```

There are obvious downsides:

- Layout and code are mixed as designers and developers can not work independently from each other.
- Not extendable (e.g. adding new markers).
- Unnecessary and complicated API functions.
- Control structures in templates are not possible.
- Only strings and arrays are supported – objects are not.

In order to address these issues, the Fluid development started with the following properties:

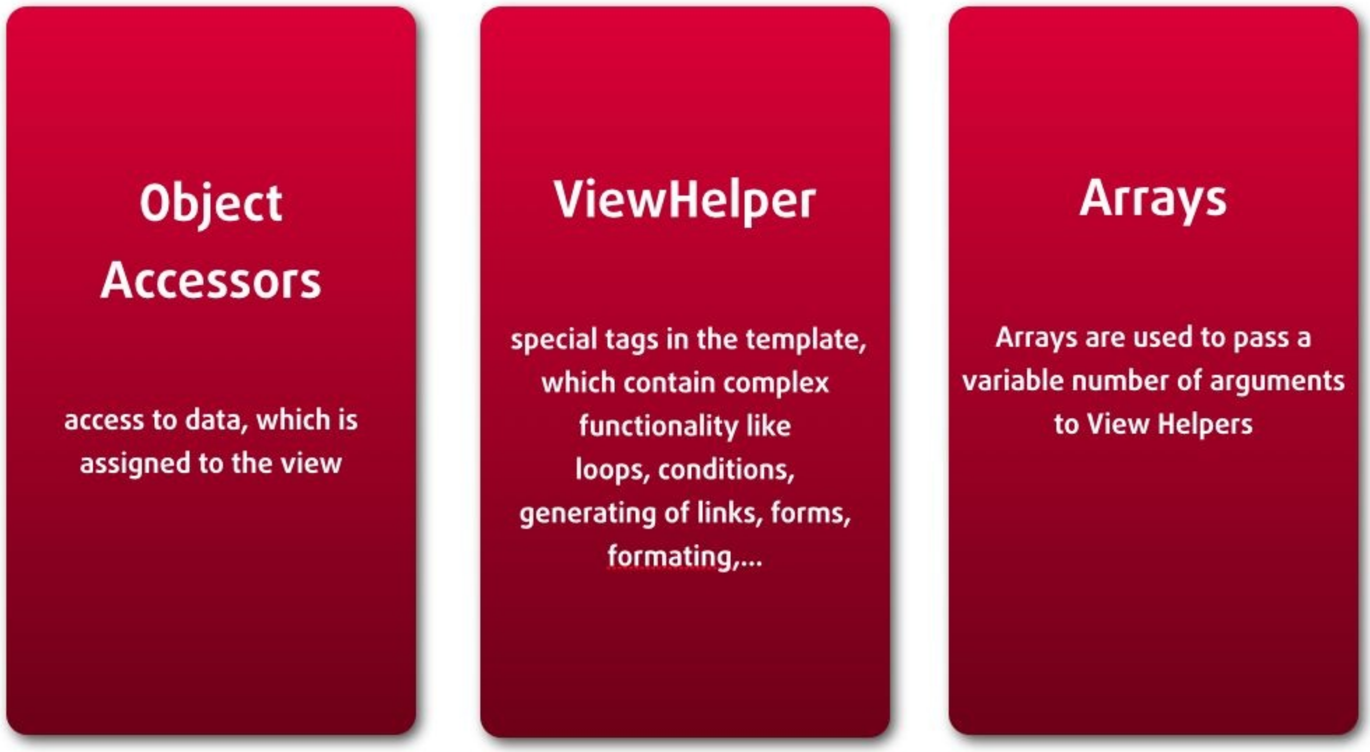
- simple and nifty template engine
- supporting the template author (e.g. auto-completion in IDE, etc.)
- easy to extend
- intuitive usage
- various output formats possible
- completely object-orientated

Additionally, one of the main requirements of Fluid is to have all logic, which has something to do with the view, exactly located there – clearly encapsulated so that no PHP code appears in HTML (which is the case in other template engines such as Smarty).

Three basic Fluid concepts exist:

- Object Accessors
- ViewHelper
- Arrays

Figure 7.5. Basic Fluid concepts


Object Accessors
access to data, which is assigned to the view
ViewHelper
special tags in the template, which contain complex functionality like loops, conditions, generating of links, forms, formating,...
Arrays
Arrays are used to pass a variable number of arguments to View Helpers

We already discussed the Object Accessor: this is the option to access values via curly brackets `{}`. As soon as a value `$value` is assigned to an identifier `$identifier` in the controller, we are able to access the value by using `{value}`.

The value can be a string, an array (numeric or associative), an object or a combination of all of these. The dot is used to access a value in a sub-level, for example a numeric array `{identifier.0}`, an associative array `{identifier.key}` or an object `{identifier.property}`.

In the case that an access to an object is made, we automatically gain access to all object properties. These properties are determined directly (if they are `public`) or via the getter methods (if they are `protected`) – for example `getTitle()` of the Blog object to provide `{blog.title}`).

In addition, this can also be used to access objects which are stored as properties: `{blogs.0.posts.1.author.lastName}`.

ViewHelper are PHP classes eventually, which implement more complex functionality. They can be accessed with tags in templates:

```
<h1>{blogTitle}</h1>

<f:if condition="{blog.posts}">
   <f:then>
      <ul>
         <f:for each="{blog.posts}" as="post">
            <li>{post.title}</li>
         </f:for>
      </ul>
   </f:then>
   <f:else>
      <p>There are no posts!<p>
   </f:else>
</f:if>
```

This example uses the *IfViewHelper*. It first checks in attribute `condition`, if object `{blog.post}` has any content. If this is the case (*ThenViewHelper*), an `<ul>...</ul>` tag is written. Inside this tag, the *ForViewHelper* iterates all elements (due to the fact that a number of Blog posts may exist) and wraps every title of a post in `<li>...</li>` tags. If object `{blog.post}` is empty, the *ElseViewHelper* part becomes relevant and the message appears that no posts exist.

ViewHelper always have the same structure:

- Syntax:

  `<f:ViewHelperName arguments>CONTENT</f:ViewHelperName>`

  or

  `<f:ViewHelperName arguments />`

- The `f:` specifies the Fluid-specific name space

  `{namespace f = TYPO3\CMS\Fluid\ViewHelpers}`

  This declaration is placed in every template and therefore the default name space of Fluid.

- All ViewHelper are based on classes.
- The file names of these classes are:

  `ViewHelperName + ViewHelper.php`

  For example: `IfViewHelper.php` of ViewHelper `<f:if condition>...</f:if>`

- The directory of all Fluid-specific ViewHelpers is: `typo3/sysext/fluid/Classes/ViewHelpers/`.
- A dot in ViewHelper denotes sub-directories, for example: `<f:format.nl2br>...</f:format.nl2br>` would result in a file `typo3/sysext/fluid/Classes/ViewHelpers/Format/Nl2brViewHelper.php`.

Multiple dots are also possible, which would result in a deeper directory structure.

At the time of this writing, 82 ViewHelpers are shipped with Fluid:

- Formatting (`format.xxx`)
- Translation (`translate`)
- Form creation (`form` and `form.xxx`)
- Link creation (`link.xxx` and `uri.xxx`)
- Backend (`be.xxx.yyy`)
- TypoScript (`cObject`)
- Control structures (`if`, `then`, `else`, `for`, `switch`, `groupedFor`, `cycle`,…)
- Layout and Partials (`render`, `section`)
- Debugging (`debug`)
- Image processing (`image`)
- Miscellaneous (`base`, `count`,…)

There are also many ViewHelpers developed by the community available. For example:

- GoogleMaps (shows information at Google Maps)
- File Extension (based on the extension of the file name, the appropriate icon is shown, e.g. PDF or DOCX)
- Smartphone Link (generates a link in a format that mobile phones dial the phone number on tap)
- Include (includes CSS or JavaScript files)

Unfortunately, at this point in time there is no official ViewHelper repository which would allow developers to search for or browse through ViewHelpers.[28] It is recommended, to collect and archive ViewHelpers which you have found or written in order to have them at hand in the future.

The TYPO3 extension `vhs` features a comprehensive library of ViewHelper, which is worth reviewing before writing your own.[29]

The last Fluid concept is named *arrays*: it is possible to pass data structures in JSON array format in Fluid. The example below shows this in attribute `arguments`:

```
<f:link.action controller="Post"
action="show" arguments="{post: currentPost,
blog: blog}">Show current post</f:link.action>
```

The following constructs are supported:

```
{ key1: 'Hello',
  key2: "World",
  key3: 20,
  key4: blog,
  key5: blog.title,
  key6: '{firstname} {post.lastname}'
}
```

Inside curly brackets, key and value pairs are stated:

`key1`
: value is a string in single quotes.

`key2`
: value is a string in double quotes.

`key3`
: value is a number

`key4`
: value is an object

`key5`
: value is a property of an object (could be a property or an object)

`key6`
: value is a string, where variables are replaced accordingly (identifier or properties)

A ViewHelper can also be used as a value. However this is only possible by using inline syntax, which we will describe in more detail later. This notation requires quotes.

Form Syntax

We just learnt the basics of Fluid and can now start building our form. Create a new file `AddForm.html` in directory `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog` and add the following content:

```
<h1>Create a new Blog</h1>

<f:form action="add" object="{blog}" name="blog" additionalAttributes="{role

    <div class="form-group">
        <label>Blog Title</label>
        <f:form.textfield property="title" class="form-control" />
    </div>

    <div class="form-group">
        <label>Blog Description</label>
        <f:form.textarea property="description" class="form-control" />
    </div>

    <f:form.submit value="Create Blog!" class="btn btn-primary" />

</f:form>
```

We are using the ViewHelper `<f:form>` which generate the form. The value `add` at attribute `action` makes sure, `addAction` of our Blog controller is called after submitting the form. `object="{blog}"` tells Extbase that the form belongs to object `blog` (which we assigned by stating `$this->view->assign()`). This also makes it easier to map the fields later. By using `name="blog"` we ensure, that the form uses exactly this name (which will use a name space later) and Extbase picks it up correctly. The last attribute `additionalAttributes` adds some specific attributes to the HTML form tag. In this case, `{role:'form'}` generates an attribute `role="form"`, which instructs Twitter Bootstrap how to style the form.

Let’s have a closer look at every single field:

Title
: We are using the `<f:form.textfield>` ViewHelper to render an input field. The important point is, that the attribute `property="title"` maps the property title of the domain object `blog`, which has been defined in the `<f:form>` ViewHelper further up. Everything else is handled by Extbase. The class is used for the visual appearance via Bootstrap.

Description
: This is similar as above, but a text area is required and therefore we are using the `<f:form.textarea>` ViewHelper instead.

Submit Button
: In order to style the submit button properly, we are using a few Bootstrap classes, namely `btn` (to convert the `<input>` field into a button) and `btn-primary` to colour it blue.

Generated HTML Form

Based on the ViewHelpers, Extbase generates a HTML markup, where we can find several interesting things:

```
<form role="form" name="blog" action="index.php?id=1&amp; tx_simpleblog_blog
<div>
<input type="hidden" name="tx_simpleblog_bloglisting[__referrer][@extension]
<input type="hidden" name="tx_simpleblog_bloglisting[__referrer][@controller
<input type="hidden" name="tx_simpleblog_bloglisting[__referrer][@action]" v
<input type="hidden" name="tx_simpleblog_bloglisting[__referrer][arguments]"
<input type="hidden" name="tx_simpleblog_bloglisting[__trustedProperties]" v
</div>

<div class="form-group">
   <label>Blog Title</label>
   <input class="form-control" type="text" name="tx_simpleblog_bloglisting[b
</div>

<div class="form-group">
   <label>Blog Description</label>
   <textarea class="form-control" name="tx_simpleblog_bloglisting[blog][desc
</div>

<input class="btn btn-primary" type="submit" name="" value="Create Blog!" />

</form>
```

Particularly interesting are the *hidden* fields at the beginning of the form. Due to the fact that the HTTP protocol is connectionless, the exact source of a POST request is unknown to the instance, that receives the request (the optional referrer string sent by the user's browser disregarded).

The hidden fields contain information about the extension, the controller and the action in order to clearly determine the origin. This is important, in particularly, if `addAction()` should validate the receiving object and – e.g. due to a failed validation – reject it. In this case, we need to come back to the origin, the `addFormAction()`.

The next two fields act as a protection against CSRF (Cross-Site Request Forgery). An attacker could manipulate the form and submit it with adulterated data from his server. This would be problematic, if there was a `user` object, which allowed users to edit their details. An *admin* flag must not be edited by the user itself of course but by an administrator user only. An attacker could simply add a checkbox *Admin*, tick it and submit the form. A CSRF protection detects this attempt, rejects the request and outputs an error message.

It's also interesting to note that the attribute `property` in the ViewHelper has been converted into the attribute `name` and expanded with a name space, e.g. `name="tx_simpleblog_bloglisting[blog][description]"`. As a result, the *Property Mapper* recognises which properties belong to which objects.

Form Processing

Now we adjust the BlogController because `addAction()` should not create the objects itself by using `new()` any more but instead use the object from `addFormAction()`.

```
/**
 * add action - adds a blog to the repository
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 */
public function addAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog) {
   $this->blogRepository->add($blog);
   $this->redirect('list');
}
```

We have created an object in `addFormAction()` in the form and passed it to `addAction()` through a POST request. As soon as `addAction()` is executed, Extbase detects that parameter `$blog` is the same as the attribute `name="blog"` in the form (by analysing the annotation as well as the method parameter). This way, Extbase reconstitutes the object `$blog` and assigns the properties passed in the request to it. The object is added to the BlogRepository and after that, the action redirects to the list view.

We also have to update the link at the end of the template `List.html` because it should no longer point to `addAction()` but to `addFormAction()`.

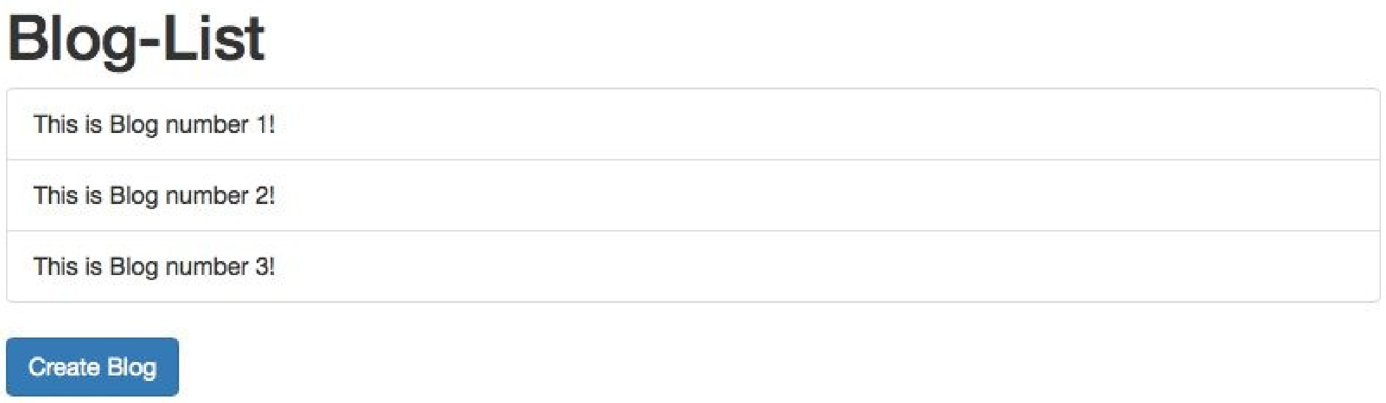
```
<f:link.action action="addForm" class="btn btn-primary">Create Blog</f:link.
```

Now we are able to create objects in a form and store them in a repository.

Pay attention to the cache

To avoid error messages, you have to clear the cache during development from time to time. For example, after updating `ext_localconf.php`, the configuration cache must be emptied. Extbase also stores code (e.g. actions) to the caching framework `cf_extbase_objects`, even if “do not cache” has been set in `ext_localconf.php`. Thus the reflection possibly accesses outdated class information. The only solution is to *clear all caches* in the backend.

Figure 7.6. List of Blogs



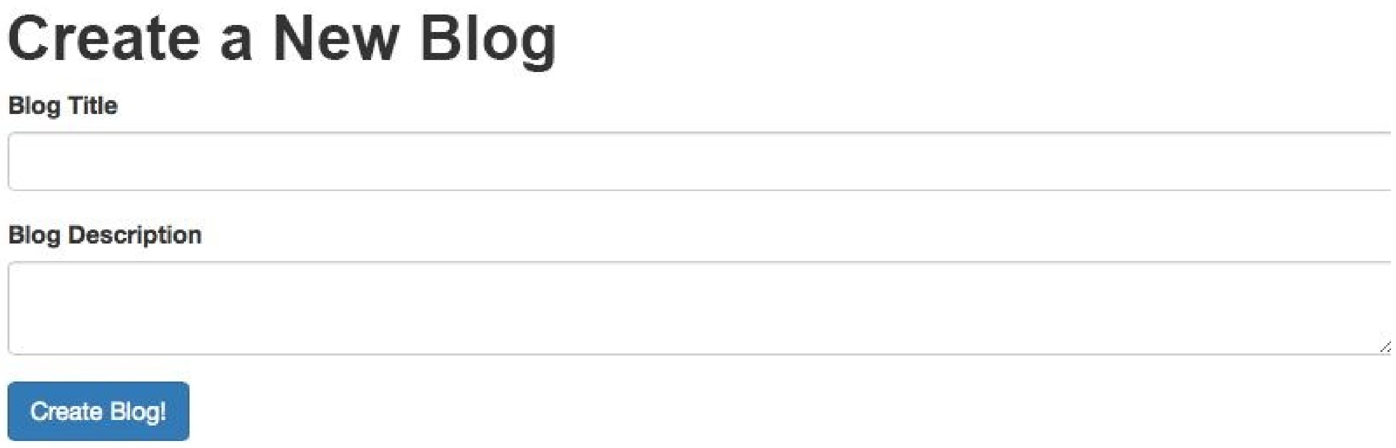
Blog-List

This is Blog number 1!

This is Blog number 2!

This is Blog number 3!

Figure 7.7. Form to create a Blog



Create a New Blog

Blog Title

Blog Description

7.2. Display a Blog (Read)

We have now reached the read state in the CRUD process – the retrieval of an object. First, we have to extend the action list in file `ext_localconf.php`, as before. The new action is named `show`:

```
...
   array(
      'Blog' => 'list, addForm, add, show',
   ),
   // non-cacheable actions
   array(
      'Blog' => 'list, addForm, add, show',
   )
...
```

In the next step we update the template `List.html` and place a button after each Blog title to allow users to access the single view of an object:

```
<ul class="list-group">
   <f:for each="{blogs}" as="blog">
      <li class="list-group-item">{blog.title} <f:link.action action="show"
   </f:for>
</ul>
```

We added the `<f:link.action>` ViewHelper, which points to the action `show`. The action needs to know which object we would like to access, so this information has to be passed somehow. This is done by the attribute `arguments`, which allows us to pass arbitrary data in an array syntax to the link target. The value `{blog:blog}` makes sure that the Blog `blog` (this is the right part) is passed via a GET request to the action under the name `blog` (this is the left part).

The classes create buttons, which are Bootstrap conform: `btn btn-primary btn-xs` results in a small, blue button and `pull-right` positions it far right.

The link could look like the following:

```
http://www.example.com/index.php?id=1&
   tx_simpleblog_bloglisting[blog]=25&
   tx_simpleblog_bloglisting[action]=add&
   tx_simpleblog_bloglisting[controller]=Blog&
   cHash=3d6a8fa3884f23198e0cb9a1604abccb
```

Through the parameter `tx_simpleblog_bloglisting[blog]=25`, Extbase knows that the Blog with the UID `25` is accessed. Actually the identification of the Blog is completely in

Extbase’s control. At the moment, UID is used. However you can always decide to use UUID (Universally Unique Identifier) instead, like in TYPO3 Flow.

The link above points to the action `show` so we have to implement the `showAction()` in BlogController in the next step:

```
/**
 * show action - displays a single blog
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 */
public function showAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog)
   $this->view->assign('blog',$blog);
}
```

The action is clear, by stating the name `blog` in the GET parameter of the link, Extbase recognises this by taking the combination of annotation and type hint into account. Extbase then retrieves the Blog with the appropriate identifier from the repository (the value of the UID has been transmitted in the request) and assigns the object to the view by `$this->view->assign()`.

Finally, we have to implement the view and to do so, create a new file `Show.html` in directory `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog` with the following content:

```
<h1>Show Blog</h1>

<dl class="dl-horizontal">
    <dt>Blog Title:</dt>
    <dd>{blog.title}</dd>
    <dt>Blog Description:</dt>
    <dd>{blog.description}</dd>
</dl>

<f:link.action action="list" class="btn btn-primary">Back to Blog listing</f
```

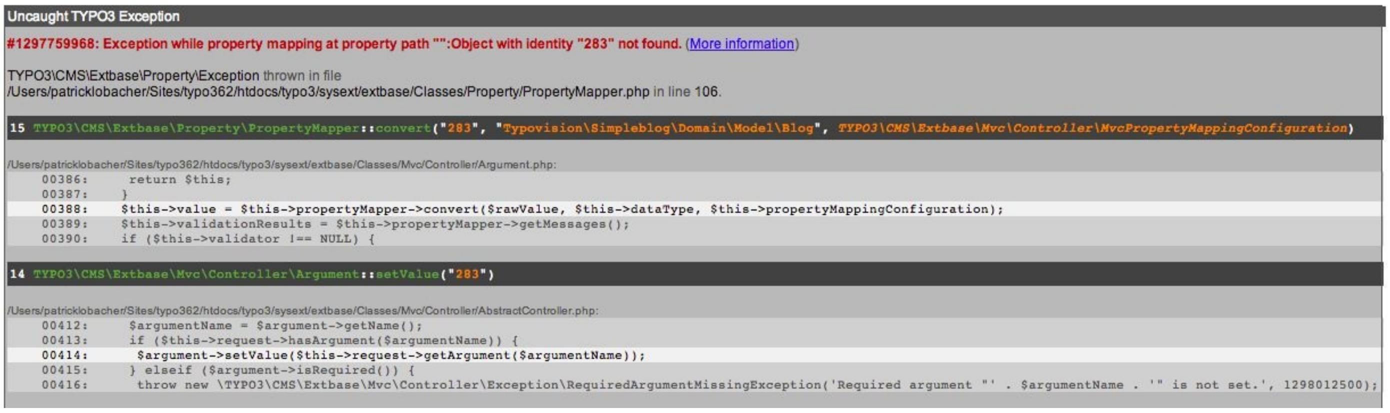
The output of the Blog happens by accessing the properties of object `{blog}`, which has been assigned in `showAction()` by `$this->view->assign()`.

The link to the `listAction()` at the end lets us jump back to the list view.

Extbase automatically retrieved the right Blog from the repository for us, without the need to implement any internal details, which are required for accessing the object.

The `cHash` parameter at the end of the GET link prevents the manipulation of the link. If you omit this parameter and change `tx_simpleblog_bloglisting[blog]` to a value that does not exist, Extbase returns an error message:

Figure 7.8. Manipulated URL results in an error message

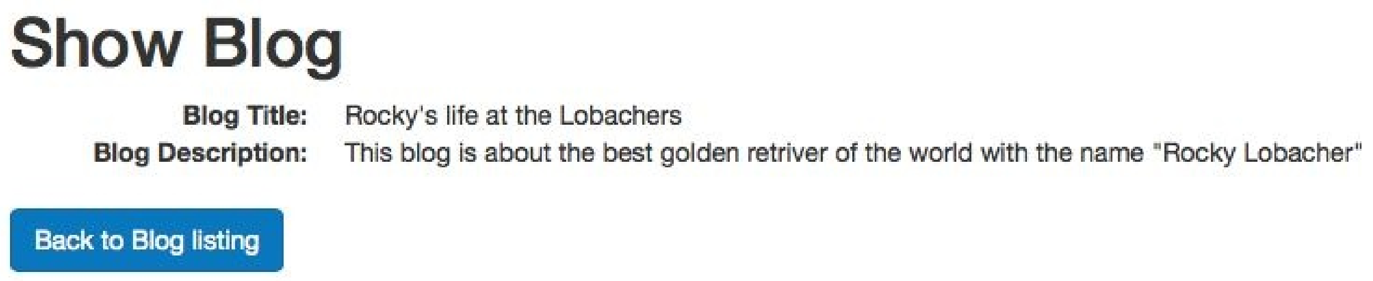


```
Uncaught TYPO3 Exception
#1297759968: Exception while property mapping at property path "":Object with identity "283" not found. (More information)

TYPO3\CMS\Extbase\Property\Exception thrown in file
/Users/patricklobacher/Sites/typo362/htdocs/typo3/sysext/extbase/Classes/Property/PropertyMapper.php in line 106.

15 TYPO3\CMS\Extbase\Property\PropertyMapper::convert("283", "Typovision\Simpleblog\Domain\Model\Blog", TYPO3\CMS\Extbase\Mvc\Controller\MvcPropertyMappingConfiguration)

/Users/patricklobacher/Sites/typo362/htdocs/typo3/sysext/extbase/Classes/Mvc/Controller/Argument.php:
    00386:      return $this;
    00387:     }
    00388:     $this->value = $this->propertyMapper->convert($rawValue, $this->dataType, $this->propertyMappingConfiguration);
    00389:     $this->validationResults = $this->propertyMapper->getMessages();
    00390:     if ($this->validator !== NULL) {

14 TYPO3\CMS\Extbase\Mvc\Controller\Argument::setValue("283")

/Users/patricklobacher/Sites/typo362/htdocs/typo3/sysext/extbase/Classes/Mvc/Controller/AbstractController.php:
    00412:      $argumentName = $argument->getName();
    00413:      if ($this->request->hasArgument($argumentName)) {
    00414:       $argument->setValue($this->request->getArgument($argumentName));
    00415:      } elseif ($argument->isRequired()) {
    00416:       throw new \TYPO3\CMS\Extbase\Mvc\Controller\Exception\RequiredArgumentMissingException('Required argument "' . $argumentName . '" is not set.', 1298012500);
```

Changing the parameter to a value that exists, results in the display of the appropriate Blog.

We completed the *read* step of the CRUD process now and can move on to the next step. Do not forget to *clear all caches*, before reproducing the steps described above.

Figure 7.9. Display of the Blog



Show Blog

Blog Title:	Rocky's life at the Lobachers
Blog Description:	This blog is about the best golden retriver of the world with the name "Rocky Lobacher"

7.3. Update an Object

The update of an object in the CRUD process is very similar to the creation of an object discussed earlier. We need a two-step action and a form again, which shows the object which we would like to update.

Let’s start with the file `ext_localconf.php`, where we add the actions `updateForm` and `update` to.

```
...
   array(
      'Blog' => 'list, addForm, add, show, updateForm, update',
   ),
   // non-cacheable actions
   array(
      'Blog' => 'list, addForm, add, show, updateForm, update',
   )
...
```

Then, continue with the BlogController:

```
   /**
    * updateForm action - displays a form for editing a blog
    *
    * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
    */
   public function updateFormAction(\Lobacher\Simpleblog\Domain\Model\Blog $
      $this->view->assign('blog',$blog);
   }

   /**
    * update action - updates a blog in the repository
    *
    * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
    */
   public function updateAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog
      $this->blogRepository->update($blog);
      $this->redirect('list');
   }
```

As before, the `updateFormAction()` “accepts” the object via the annotation and the type hint and passes it on to the view.

As soon as the view (which we will amend later) sends the updated object back to the `updateAction()`, merely method `update()` is executed on the repository. Extbase takes care of finding the object, which should be updated, as well as of the update and the persistence. After that, the user will be redirected back to the list of Blogs.

Let's have a look at the view, for which we create a file `UpdateForm.html` in directory `typo3conf/ext/simpleblog/Resources/Private/Templates` with the following content:

```
<h1>Edit Blog</h1>

<f:form action="update" object="{blog}" name="blog" additionalAttributes="{r

   <div class="form-group">
      <label>Blog Title</label>
      <f:form.textfield property="title" class="form-control" />
   </div>

   <div class="form-group">
      <label>Blog Description</label>
      <f:form.textarea property="description" class="form-control" />
   </div>

   <f:form.submit value="Update Blog!" class="btn btn-primary" />

</f:form>
```

Strictly speaking, this code is almost identical with the code for creating an object, except two labels and the actions are different. Before you come up with the idea to optimise this by using one form only and differentiate between those two programmatically, wait until we explained "Partials" – these are much more suited for this kind of approach.

In order to be able to access the update form, we have to add a button to the list template (`List.html`):

```
...
<f:link.action action="updateForm" arguments="{blog:blog}" class="btn btn-pr
<f:link.action action="show" arguments="{blog:blog}" class="btn btn-primary
...
```

You already know the lower of the two links (only an additional class `margin-right` has been added). The upper one leads to the `updateFormAction()` and attribute `arguments` transfers the current Blog into it.

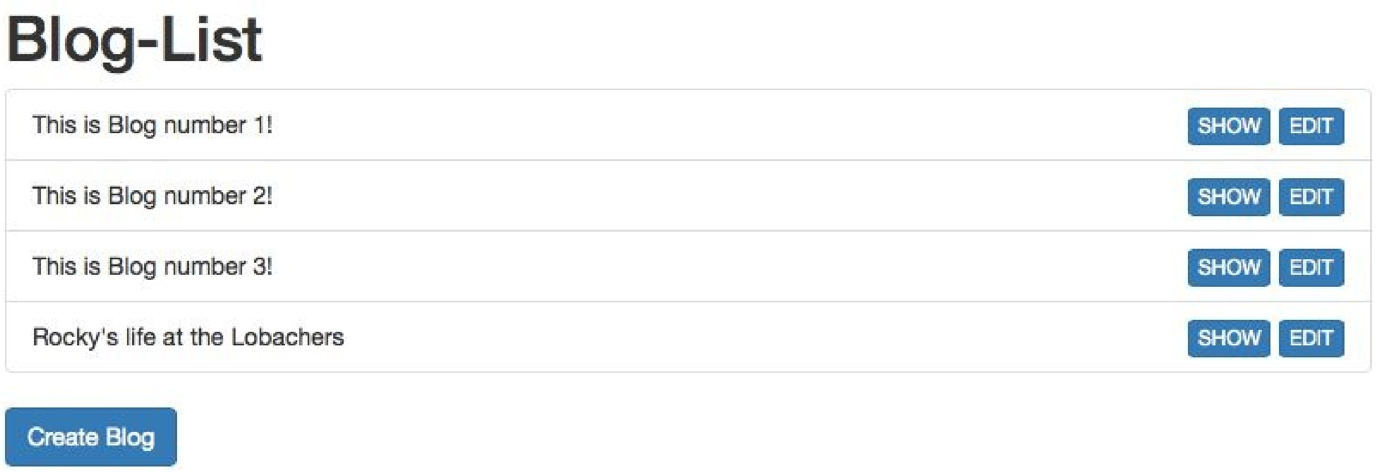
For an improved visual look (the buttons should not stick together too closely), we add the following CSS code to file `typo3conf/ext/simpleblog/Resources/Public/Css/simpleblog.css`:

```
.margin-right {
    margin-right: 5px;
}
```

Do not forget to delete the cache (*clear all caches*) before you test the new function.

If you click the *EDIT* button now, you will get the form with the object data filled. This means, Extbase accessed the repository in the background, retrieved the object and populated the form fields appropriately.

Figure 7.10. List of Blogs with edit button



7.4. Deletion of an Object

Deleting an object is simple. As before, we first add the action `delete` in `file ext_localconf.php`.

```
...
   array(
      'Blog' => 'list, addForm, add, show, updateForm, update, delete',
   ),
   // non-cacheable actions
   array(
      'Blog' => 'list, addForm, add, show, updateForm, update, delete',
   )
...
```

Then, the method `deleteAction()` in BlogController is required:

```
   /**
    * delete action - deletes a blog in the repository
    *
    * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
    */
   public function deleteAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog
      $this->blogRepository->remove($blog);
      $this->redirect('list');
   }
```

The only new thing is, that the method to delete an object from the repository is called `remove()`.

In theory, we do not need a View at all, because we could simply delete the Blog directly. However a confirmation would be useful so we will implement a View, which allows us to do so.

Please try to develop the action and the template yourself, so that the user has to confirm the deletion of the Blog. After that, compare your solution with the example code below.

First, we have to extend the list of available actions in file `ext_localconf.php` by `deleteConfirm` (or a similar name):

```
...
   array(
      'Blog' => 'list, addForm, add, show, updateForm, update, deleteConfirm
   ),
   // non-cacheable actions
   array(
      'Blog' => 'list, addForm, add, show, updateForm, update, deleteConfirm
   )
```

```
...
```

This is followed by the action `deleteConfirmAction()` in BlogController:

```
   /**
    * deleteConfirm action - displays a form for confirming the deletion of
    *
    * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
    */
   public function deleteConfirmAction(\Lobacher\Simpleblog\Domain\Model\Blo
      $this->view->assign('blog',$blog);
   }
```

And finally a template `DeleteConfirm.html` in directory `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog`:

```
<h1>Delete Blog?</h1>

Are you sure you want to delete the Blog <strong>{blog.title}</strong>?

<br /><br />

<f:link.action action="list" class="btn btn-danger">Cancel!</f:link.action>
<f:link.action action="delete" class="btn btn-success" arguments="{blog:blog
```

If the user chooses not to delete the Blog, we simply redirect him back to the `listAction()`. If the user confirms the deletion of the Blog, we append the Blog object to the link by using `arguments`. This way, `deleteAction()` knows which Blog should be removed from the repository.

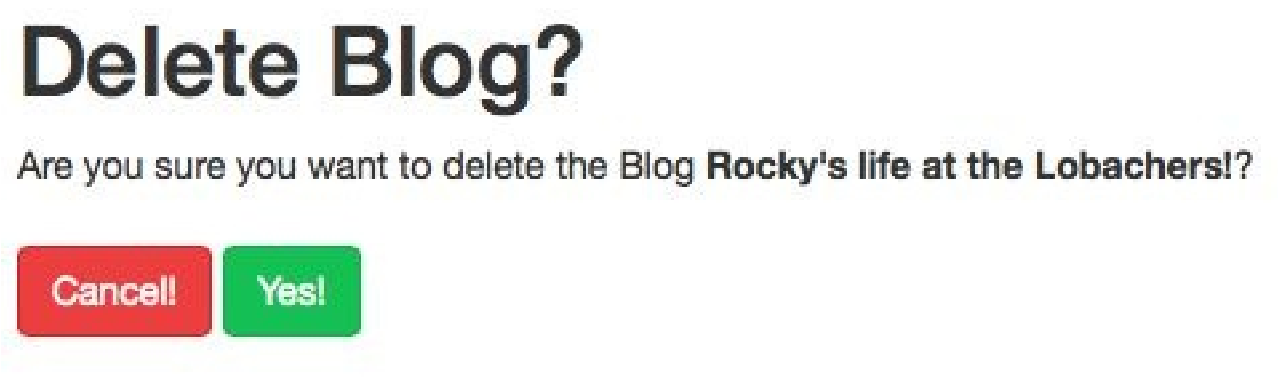
Likewise, we must not forget to add the link to the Blog list, which allows us to redirect to the `deleteConfirmAction()`:

```
...
<f:link.action action="deleteConfirm" arguments="{blog:blog}" class="btn btn
<f:link.action action="updateForm" arguments="{blog:blog}" class="btn btn-pr
...
```

In addition, CSS class `margin-right` at the edit link adds a space between the two buttons.

Last but not least, do not forget to *clear all caches* before testing your work!

Figure 7.11. Confirmation of a Blog deletion


Delete Blog?

Are you sure you want to delete the Blog **Rocky's life at the Lobachers!**?

---

[27] Further details at http://typo3.org/documentation/article/the-mysteries-of-chash-1.

[28] Forge (http://forge.typo3.org/search?q=viewhelper&scope=all&all_words=1&issues=1&submit=Submit) or Git (http://git.typo3.org) may reveal some existing solutions.

[29] https://fluidtypo3.org/viewhelpers/vhs/master.html; also available in the TER at http://typo3.org/extensions/repository/view/vhs

Chapter 8. Fluid Templating: Templates, Layouts and Partials

Fluid provides a flexible concept in order to meet all possible layout requirements.

The following three elements are available to achieve this:

Templates
: Templates are HTML files, which are loaded automatically by the framework. For example: the file `.../Resources/Private/Templates/Blog/List.html` is loaded by the Controller `Blog` in the action `list`. Inside templates there may be sections, which are rendered by the ViewHelper `<f:section>` in templates and `<f:render>` in layouts.

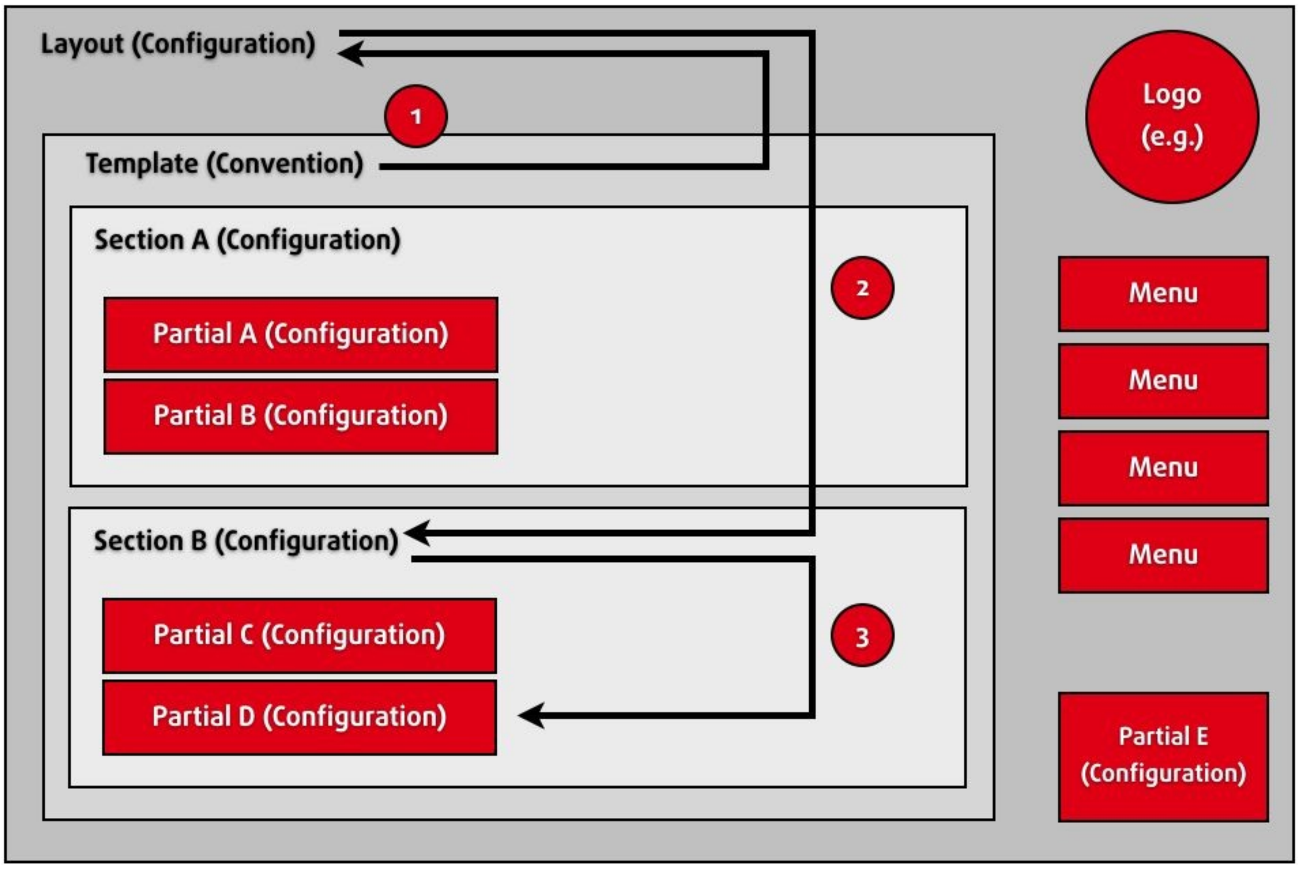
Layouts
: Layouts are used for global styling, e.g. if you want to maintain a logo or functional menu across the entire site. ViewHelper `<f:layout>` is responsible for including layouts in templates.

Partials
: Partials are small template units, which are perfect to fulfil recurring tasks, e.g. the output of information sets to multiple parts of your layout. Like layouts, partials can also be configured and the ViewHelper `<f:render>` handles this.

The process of a template rendering works as follows:

1. As per convention, the template name is determined: `Resources/Private/Templates/[Controller]/[ActionName].html`. If the template file does not contain any LayoutViewHelper `<f:layout>`, the rendering of the template is complete. Where a LayoutViewHelper `<f:layout>` exists, the appropriate layout is loaded and rendered. The layout must contain at least a RenderViewHelper `<f:render section="...">`.

Figure 8.1. Template processing in Fluid


Layout (Configuration)
1
Template (Convention)
Logo
(e.g.)
Section A (Configuration)
2
Partial A (Configuration)
Partial B (Configuration)
Menu
Menu
Menu
Menu
Section B (Configuration)
Partial C (Configuration)
3
Partial D (Configuration)
Partial E
(Configuration)

2. This section is now determined in the Template through the existence of the line `<f:section name="...">`.
3. Inside this section (or at an arbitrary position in the Layout), layouts (partials) referenced by `<f:render partial="...">` can be included.

8.1. Creating and Referencing Layouts

Assuming, we would like to define a layout for our application, we have to create a folder `typo3conf/ext/simpleblog/Resources/Private/Layouts` and a file `Default.html` inside this folder. This file contains the following code:

```
<div class="layout1">

<f:render section="content" />

</div>

&copy; 2015 by <a href="http://www.lobacher.de">LOBACHER.</a>
```

This example shows a typical layout. Our intention is to draw a box around the main application (CSS class `layout1`) and output a copyright notice.

The ViewHelper `<f:render section="content" />` reads the template and searches for a section named `content` (note: we have to update *all* templates if we want to use this Layout in every template).

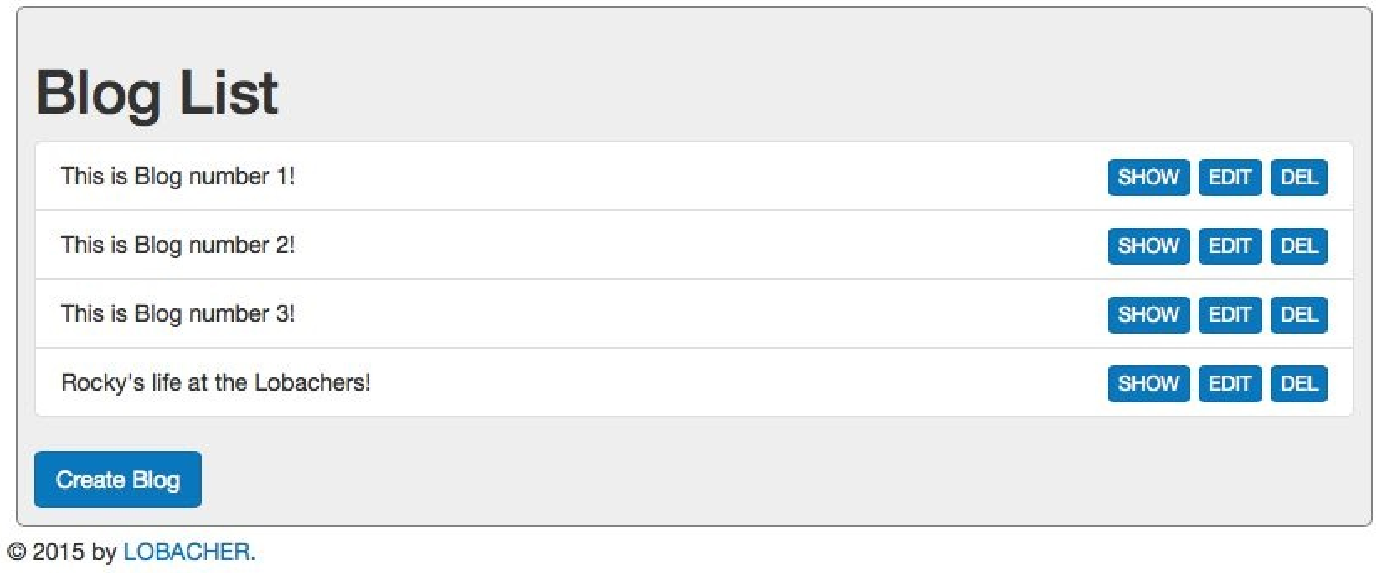
Please see template `List.html` below (and add the lines to all templates):

```
<f:layout name="default" />

<f:section name="content">

<h1>Blog List</h1>
...
<f:link.action action="addForm" class="btn btn-primary">Create Blog</f:link.

</f:section>
```

Finally, we should add a directive to the CSS file `simpleblog.css`:

```
.layout1 {
    background-color: #eee;
    padding:10px;
    margin:5px;
    border: 1px solid grey;
    -webkit-border-radius: 5px;
    -moz-border-radius: 5px;
    border-radius: 5px;
}
```

Figure 8.2. Output of the layout


Blog List

This is Blog number 1!

This is Blog number 2!

This is Blog number 3!

Rocky's life at the Lobachers!

© 2015 by LOBACHER.

It is possible to define multiple sections in a template. For example, you could have several layouts, which include different sections in the same template.

In theory, you could also include sections recursively, see the example below.

```
<f:section name="mySection">
 <ul>
   <f:for each="{myMenu}" as="menuItem">
     <li>
       {menuItem.text}
       <f:if condition="{menuItem.subItems}">
         <f:render section="mySection" arguments="{myMenu: menuItem.subItems
       </f:if>
     </li>
   </f:for>
 </ul>
</f:section>
<f:render section="mySection" arguments="{myMenu: menu}" />
```

This would result in the following output, depending on the content of `{myMenu}`:

```
<ul>
  <li>menu1
    <ul>
      <li>menu1a</li>
      <li>menu1b</li>
    </ul>
  </li>
[...]
```

8.2. Partials

Partials are “small(er)” template sections, which can be re-used. This allows you to include content from one source rendered by a ViewHelper in multiple places.

8.2.1. Simple Partials

We are so happy with our Blog, we definitely need a Twitter button to allow users to promote it in their timeline.

To build our Twitter button, we have to create a directory `Partials` inside `typo3conf/ext/simpleblog/Resources/Private/`. The partial with the name `TwitterShare.html` should show the following content:

```
<br /><br /><a href="https://twitter.com/share" class="twitter-share-button"
<script>!function(d,s,id){var js,fjs=d.getElementsByTagName(s)[0]; if(!d.get
```

The code above is based on the example at https://dev.twitter.com/docs/tweet-button, “Ways to add the Tweet Button to your website”, but to avoid using old code, please check the documentation for an up-to-date implementation of the button.

The last step is to add the partial to the page, which can be done either by including it in the template or by including it at the end of the layout (easier).

```
...
&copy; 2015 by <a href="http://www.lobacher.de">LOBACHER.</a>

<f:render partial="TwitterShare" />
```

Using sub-directories for partials and layouts and file extensions

Partials as well as layouts can be stored in sub-directories. For example, `<f:render partial="Social/TwitterShare" />` would search for a file `TwitterShare.html` in path `.../Resources/Private/Partials/Social`.

You might wonder why we omitted the suffix `.html`: the reason for this is that we would like to keep the format “flexible”. During template-rendering, the system uses `*.html` by default, however, you can configure this globally and change the suffix to another format, such as `*.xml` for example. If we then want to address `TwitterShare.html`, we would have to append `.html` to the name.

8.2.2. Complex Partials

Let's take full advantage of the partial mechanism to make both forms (create and update an object) more consistent. Our two labels currently require two forms, so it seems ideal to maintain just one form and to keep the varying components outside, included as partials.

Let's get started and create a new folder `Blog` in `typo3conf/ext/simpleblog/Resources/Private/Partials` in order to separate all partials, which belong to one object. After that, we will create a file `Form.html` inside the Blog directory and copy a specific part of the template `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/AddForm.html`:

```
<h1>Create New Blog</h1>

<f:form action="add" object="{blog}" name="blog" additionalAttributes="{role

   <div class="form-group">
      <label>Blog Title</label>
      <f:form.textfield property="title" class="form-control" />
   </div>

   <div class="form-group">
      <label>Blog Description</label>
      <f:form.textarea property="description" class="form-control" />
   </div>

<f:form.submit value="Create Blog!" class="btn btn-primary" />

</f:form>
```

This part needs to be removed from the template `AddForm.html` and replaced by the following line:

```
...
<f:render partial="Blog/Form" />
...
```

While this is now working, in order to be able to access the partial via action `Update`, we have to dynamically call the areas below using the "Object Accessors" in the format `{...}` inside the partial `Form.html`:

```
<h1>{headline}</h1>

<f:form action="{action}" object="{blog}" name="blog" additionalAttributes="

...

<f:form.submit value="{submitmessage}" class="btn btn-primary" />

</f:form>
```

As a result, we have the three Object Accessors: `{headline}`, `{action}` and `{submitmessage}`, which need to be filled when the partial is executed. Change the line in template `AddForm.html` as follows:

```
<f:render partial="Blog/Form" arguments="{headline:'Create New Blog',action:
```

The transfer of the values listed above is carried out by the attribute `argument` of the array syntax. The Blog is not being transferred into the partial and is currently only available in the template. We can address this, by adding `blog:blog` to the partial.

Passing arguments to partials

No values are transferred into the partial by default except settings. To make any individual value available to a partial, you have to make use of the attribute `arguments`. Alternatively, the keyword `{_all}` transfers all values into the partial.

The same needs to be done for the template `UpdateForm.html`.

Exercise

Try to implement the same logic in the template `UpdateForm.html` and compare your solution with the example below.

File `UpdateForm.html` requires the following code instead of the form:

```
<f:render partial="Blog/Form" arguments="{headline:'Edit Blog',action:'updat
```

Chapter 9. Query Manager and Repositories

We will now investigate the Repository and Query Manager in more detail.

As per convention, we will find repositories in the directory `Classes/Domain/Repository` with their names based on the domain objects for which they are responsible. So in our case, directories `BlogRepository` and `PostRepository` already exist.

9.1. Structure of a Repository Class

By default, the structure of a repository class is fairly simple:

```
<?php
namespace Lobacher\Simpleblog\Domain\Repository;

...

class BlogRepository extends \TYPO3\CMS\Extbase\Persistence\Repository {

}
?>
```

This is because many functions are already implemented in class `\TYPO3\CMS\Extbase\Persistence\Repository`, from which the class `BlogRepository` has been derived.

9.2. Repository Functions for Write Operations

For example, the following functions exist for write operations, which we have used in the CRUD process:

`add($object)`
: Adds an object to the repository.

`remove($object)`
: Removes an object from the repository.

`removeAll()`
: Removes all objects from a repository.

`update($modifiedObject)`
: Updates an object in the repository.

9.3. Repository Functions for Read Operations

Additionally, a number of functions for read operations exist, despite the fact that we have used only `findAll()` so far:

`findAll()`
: Returns all objects of a repository.

`countAll()`
: Returns the number of all objects currently stored in the repository.

`findByUid($uid)`
: Finds an object based on a UID. This is useful when only the UID is known (e.g. when using TYPO3's APIs).

Furthermore, there are *magic functions*, so-called because they do not exist in the Extbase code. When using magic function calls, Extbase checks to ensure they follow a specific structure and executes them. These functions are used to construct methods, which can be applied to properties of domain objects. For example, `findByName($value)` would be a method, which checks for matches of property `name` and `$value`. All objects matching this check are returned.

The Extbase code (`typo3/sysext/extbase/Classes/Persistence/Repository.php`) looks like this:

```
/**
 * Dispatches magic methods (findBy[Property]())
 *
 * @param string $methodName The name of the magic method
 * @param string $arguments The arguments of the magic method
 * @throws \TYPO3\CMS\Extbase\Persistence\Generic\Exception\UnsupportedMe
 * @return mixed
 * @api
 */
public function __call($methodName, $arguments) {
   if (substr($methodName, 0, 6) === 'findBy' && strlen($methodName) > 7)
      $propertyName = lcfirst(substr($methodName, 6));
      $query = $this->createQuery();
      $result = $query->matching($query->equals($propertyName, $arguments
      return $result;
   } elseif (substr($methodName, 0, 9) === 'findOneBy' && strlen($methodN
      $propertyName = lcfirst(substr($methodName, 9));
      $query = $this->createQuery();

      $result = $query->matching($query->equals($propertyName, $arguments
      if ($result instanceof \TYPO3\CMS\Extbase\Persistence\QueryResultIn
         return $result->getFirst();
      } elseif (is_array($result)) {
         return isset($result[0]) ? $result[0] : NULL;
      }

   } elseif (substr($methodName, 0, 7) === 'countBy' && strlen($methodNam
      $propertyName = lcfirst(substr($methodName, 7));
      $query = $this->createQuery();
      $result = $query->matching($query->equals($propertyName, $arguments
      return $result;
   }
   throw new \TYPO3\CMS\Extbase\Persistence\Generic\Exception\Unsupported
}
```

The following three methods use `Property` as a placeholder for an arbitrary property name in the domain model:

`findByProperty($value)`
: inds all objects, where property is equal to `$value`.

`findOneByProperty($value)`
: Returns the first object found, where `property` is equal to `$value`.

`countByProperty($value)`
: Returns the number of objects, where `property` is equal to `$value`.

9.4. Default Structure of a Query

It is not only possible to access the repository by using existing methods (or magic methods), but it is also possible to create your own methods. The name that you choose is arbitrary – but it must not start with `findBy` or `findOneBy`.

As an example, we would like to write a repository method which enables us to find a Blog that contains a specific keyword in its title.

We could use the method `findByTitle()` in the list action of the BlogController:

```
public function listAction() {
   $this->view->assign('blogs',$this->blogRepository->findByTitle('Blog'));
}
```

This should find all Blogs which contain the word `Blog` in their title. In fact, this is not the case; only Blogs where the title is exactly `Blog` are returned. We have to find a different solution by developing our own method.

Let us re-write our method call in BlogController first:

```
public function listAction() {
   $this->view->assign('blogs',$this->blogRepository->findSearchWord('Blog')
}
```

Then we create a new method `findSearchWord()` in the BlogRepository (`typo3conf/ext/simpleblog/Classes/Domain/Repository/BlogRepository.php`):

```
public function findSearchWord($search) {

   $query = $this->createQuery();

   $result = $query->execute();

   return $result;

}
```

The content of this method replicates the default behaviour of `findAll()` in that it returns all objects.

Generally speaking, the Query Manager is triggered by `$this->createQuery();` first. The second line executes the query. However this is not 100% true. In fact nothing is executed at all, but a construction plan for the query is returned. Function `execute()` allows us to pass the value `TRUE` as a parameter, which replaces the query setting `setReturnRawQuery` (which has been deprecated since TYPO3 CMS 6.2 LTS).

At this point, there are further methods to determine the result directly if desired:

`getFirst()`
: Determines the first element of the query set. We could output this via `var_dump()`, but this would result in a very large object. Try the following: `echo '<pre>' . print_r($query->getFirst(),1) . '</pre>'; die();`. We will introduce better debugging options in the next section. In order to retrieve the first object from the repository only and output it, the following could be used: `return $query->execute()->getFirst()`.

`count()`
: This method allows you to count the number of objects in a QueryResult.

`toArray()`
: This returns an array with the objects of a result set.

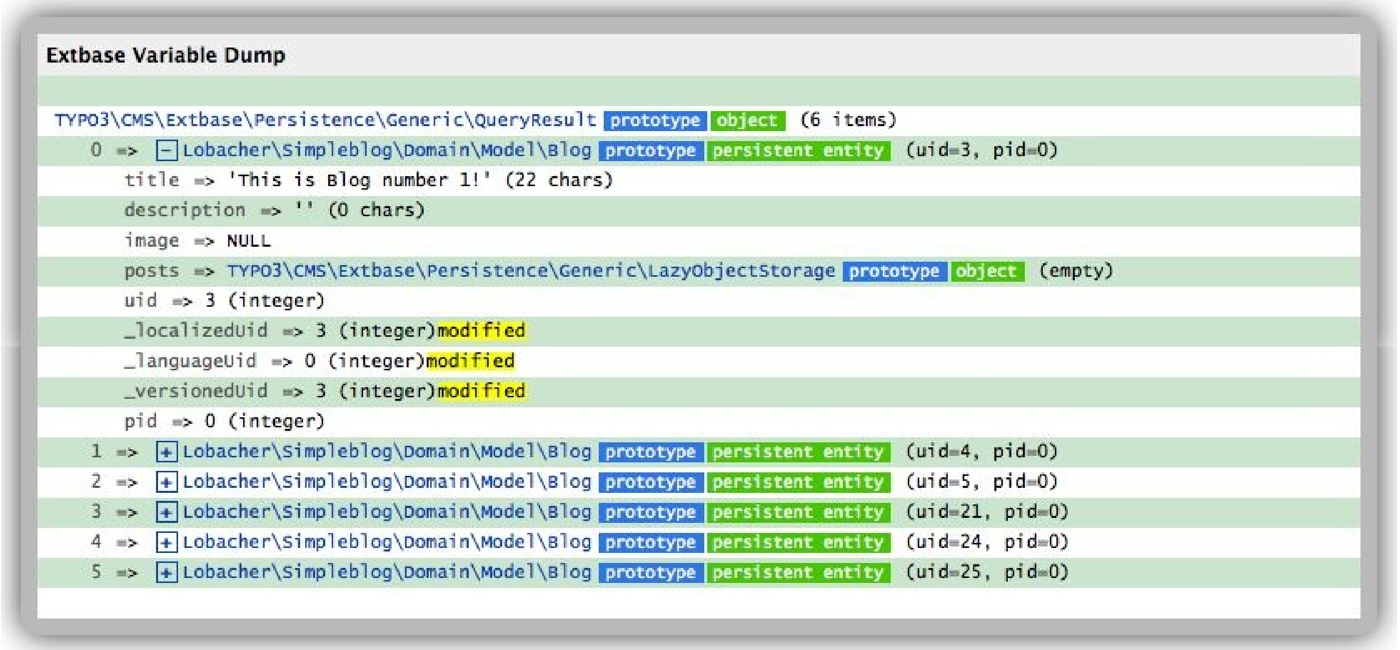
9.4.1. Side Note: Debugging

If Extbase is not working as expected, it is helpful to have more sophisticated debugging tools on hand. To achieve this, a powerful debugger has been integrated into Extbase, which originates from TYPO3 Flow.

```
public function findSearchWord($search) {
   $query = $this->createQuery();
   $result = $query->execute();

   \TYPO3\CMS\Extbase\Utility\DebuggerUtility::var_dump($result);
   die();
 }
```

The output looks very convenient, but cheats a little bit; the query is of type `TYPO3\CMS\Extbase\Persistence\Generic\QueryResult`, which usually does not contain the result (the data itself). The debugger triggers the function so that the query is applied to the repository and is therefore able to show the relevant data.

Figure 9.1. Example output of the debugger



```
Extbase Variable Dump

TYPO3\CMS\Extbase\Persistence\Generic\QueryResult prototype object  (6 items)
   0 => [-]Lobacher\Simpleblog\Domain\Model\Blog prototype persistent entity  (uid=3, pid=0)
      title => 'This is Blog number 1!' (22 chars)
      description => '' (0 chars)
      image => NULL
      posts => TYPO3\CMS\Extbase\Persistence\Generic\LazyObjectStorage prototype object  (empty)
      uid => 3 (integer)
      _localizedUid => 3 (integer)modified
      _languageUid => 0 (integer)modified
      _versionedUid => 3 (integer)modified
      pid => 0 (integer)
   1 => [+]Lobacher\Simpleblog\Domain\Model\Blog prototype persistent entity  (uid=4, pid=0)
   2 => [+]Lobacher\Simpleblog\Domain\Model\Blog prototype persistent entity  (uid=5, pid=0)
   3 => [+]Lobacher\Simpleblog\Domain\Model\Blog prototype persistent entity  (uid=21, pid=0)
   4 => [+]Lobacher\Simpleblog\Domain\Model\Blog prototype persistent entity  (uid=24, pid=0)
   5 => [+]Lobacher\Simpleblog\Domain\Model\Blog prototype persistent entity  (uid=25, pid=0)
```

In most cases, you also want to know how Extbase came to a specific result, e.g. when accessing the database you would like to see the SQL command(s) executed. This is not possible with Extbase by default, however you can trick Extbase into revealing this information.

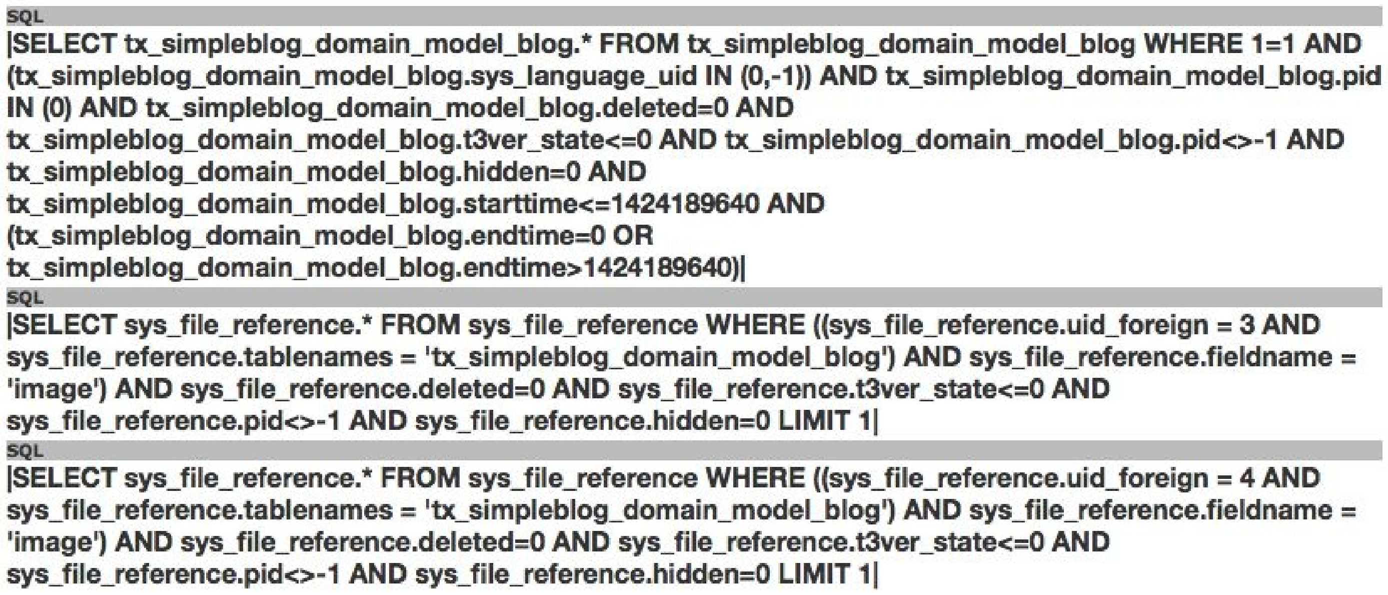
In the file `TYPO3\CMS\Extbase\Persistence\Generic\Storage\Typo3DbBackend.php` you will find the so-called storage backend of TYPO3 – the place where all SQL queries are passed to the database.

In the method `getRowsFromDatabase()` at approximately line 362, you can add a debug statement:

```
...
$this->checkSqlErrors();
debug($this->databaseHandle->debug_lastBuiltQuery,'SQL');
return $rows;
...
```

By adding the debug statement, the debug output shows a few SQL queries:

Figure 9.2. Output SQL queries



```
SQL
|SELECT tx_simpleblog_domain_model_blog.* FROM tx_simpleblog_domain_model_blog WHERE 1=1 AND
(tx_simpleblog_domain_model_blog.sys_language_uid IN (0,-1)) AND tx_simpleblog_domain_model_blog.pid
IN (0) AND tx_simpleblog_domain_model_blog.deleted=0 AND
tx_simpleblog_domain_model_blog.t3ver_state<=0 AND tx_simpleblog_domain_model_blog.pid<>-1 AND
tx_simpleblog_domain_model_blog.hidden=0 AND
tx_simpleblog_domain_model_blog.starttime<=1424189640 AND
(tx_simpleblog_domain_model_blog.endtime=0 OR
tx_simpleblog_domain_model_blog.endtime>1424189640)|
SQL
|SELECT sys_file_reference.* FROM sys_file_reference WHERE ((sys_file_reference.uid_foreign = 3 AND
sys_file_reference.tablenames = 'tx_simpleblog_domain_model_blog') AND sys_file_reference.fieldname =
'image') AND sys_file_reference.deleted=0 AND sys_file_reference.t3ver_state<=0 AND
sys_file_reference.pid<>-1 AND sys_file_reference.hidden=0 LIMIT 1|
SQL
|SELECT sys_file_reference.* FROM sys_file_reference WHERE ((sys_file_reference.uid_foreign = 4 AND
sys_file_reference.tablenames = 'tx_simpleblog_domain_model_blog') AND sys_file_reference.fieldname =
'image') AND sys_file_reference.deleted=0 AND sys_file_reference.t3ver_state<=0 AND
sys_file_reference.pid<>-1 AND sys_file_reference.hidden=0 LIMIT 1|
```

The first query determines the existing Blogs and the subsequent queries check if there are any file references.

9.5. Adjusting Queries

In general, any further selection of data and/or objects required is handled by the method `matching()`. This method allows us to initiate a limitation of records returned based on constraints.

The code in Extbase

The file responsible for the query handling in Extbase is `TYPO3\CMS\Extbase\Persistence\Generic\Query.php`.

9.5.1. Determine Result Set

`matching()` supports the following methods, which concretise the result set:

`equals($propertyName, $operand, $caseSensitive = TRUE)`
: Returns the records, whose property `$propertyName` matches operand `$operand`.

`like($propertyName, $operand, $caseSensitive = TRUE)`
: Returns the records, where operand `$operand` occurs in the property `$propertyName`. To achieve a real “LIKE”, we have to use placeholders such as `%` – for example: `'%' . $search . '%'`.

`contains($propertyName, $operand)`
: Returns a “contains” criterion which matches if the multi-valued property contains the given operand, where `$propertyName` is the name of the property to compare against and `$operand` is the value with which to compare.

`in($propertyName, $operand)`
: Returns the records where the property’s value is contained in the multi-valued operand `$operand` and `$propertyName` is the name of the property to compare against.

`lessThan($propertyName, $operand)`
: Returns records where the value of `$propertyName` is less than operand `$operand`.

`lessThanOrEqual($propertyName, $operand)`
: Returns records where the value of `$propertyName` is less than or equal operand `$operand`.

`greaterThan($propertyName, $operand)`
: Returns records where the value of `$propertyName` is greater than operand `$operand`.

`greaterThanOrEqual($propertyName, $operand)`
: Returns records where the value of `$propertyName` is greater than or equal operand `$operand`.

9.5.2. Limiting Result Set

The result set can be even limited further by using the following methods:

`setOrderings(array $orderings)`
: This method allows us to define the sort order of the results. Parameter $orderings is an array, which means Extbase sorts the result based on the first item, then second, etc. An item consists of a key (property) and the sort direction; either: `\TYPO3\CMS\Extbase\Persistence\QueryInterface::ORDER_ASCENDING` (value: `ASC`); or, `\TYPO3\CMS\Extbase\Persistence\QueryInterface::ORDER_DESCENDING` (value: `DESC`). Alternatively, you could set the protected property `$defaultOrderings` if you want to sort by a specific property in all methods.

`setLimit($limit)`
: In order to limit the return of a large number of records this method can be used. Parameter $limit must be an integer (an automatic conversion does not happen). Method `unsetLimit()` annuls the limit.

`setOffset($offset)`
: Sets the start offset to `$offset`, which means the result set skips the first records and starts at the given position, e. g. the fifth element.

9.5.3. Logical Conjunction

Additionally, we can construct logical conjunctions:

`logicalAnd($constraint1)`
: Performs a logical conjunction of the given constraints. The method takes one or more constraints and concatenates them with a boolean AND.

`logicalOr($constraint1)`
: Same as above, but with a logical OR.

`logicalNot($constraint1)`
: Same as above, but with a logical NOT.

9.5.4. Native SQL

The Query Manager enables you to construct complex queries, practically without limits. However sooner or later you might reach a point where you want to execute native SQL and even this is possible:

`statement($statement, array $parameters = array())`
: This method allows you to create native SQL statements. The query $statement may contain placeholders (`?`), which can be populated with values in array `$parameters`. `statement` also returns objects, as long as the result set contains the same fields as the properties in the domain model. Where this is not the case, the result set returned is merely an array.

9.5.5. Query Settings

Furthermore, some specific query settings exist which can be stated as follows:

```
$query->getQuerySettings()->setRespectStoragePage(FALSE);
```

Extbase currently supports the following settings:

`setRespectStoragePage($respectStoragePage)`
: Data sets are retrieved from a specific storage page by default but this behaviour can be annulled by this setting, thereby forcing Extbase to searche for the data across the entire page tree.

`setStoragePageIds(array $storagePageIds)`
: Parameter `$storagePageIds` may contain a list of all UIDs of pages, from which data sets should be read. This is also possible via TypoScript.

`setRespectSysLanguage($respectSysLanguage)`
: Typically, only data sets in the language of the currently configured language in TYPO3 are taken into account (where a language overlay exists). Setting parameter `$respectSysLanguage` to `FALSE` in this method enables you to retrieve all languages of a data set.

`setLanguageOverlayMode($languageOverlayMode)`
: This method sets the language overlay mode. Possible values of parameter `$languageOverlayMode` are `TRUE`, `FALSE` or `hideNonTranslated` are accepted.

`setLanguageMode($languageOverlayMode)`
: This method sets the language mode. Possible values of parameter `$languageMode` are `NULL`, `content_fallback`, `strict` or `ignore` are accepted.

`setLanguageUid($languageUid)`
: To set the language ID manually, this method can be used.

`setIgnoreEnableFields($ignoreEnableFields)`
: The determination of data sets follows the *Enable Fields* constraints, which means hidden records – as well as records with a start and end date not equal to the current timestamp – are not retrieved. You can override this by using the `setIgnoreEnableFields` option.

`setEnableFieldsToBeIgnored($enableFieldsToBeIgnored)`
: This method instructs Extbase to ignore certain Enable Fields (which are defined in `$GLOBALS['TCA'][$table]['ctrl']['enablecolumns']`).

`setIncludeDeleted($includeDeleted)`
: To also return deleted records, set this option to `TRUE`.

`setReturnRawQueryResult($returnRawQueryResult)`
: This value is set to `FALSE` by default, which results in result set being mapped to an object. Where this is not desirable, set the parameter `$returnRawQueryResult` to `TRUE`. This could be useful when using `statement()`. If the value is `TRUE`, you get an array containing the data from the database. It should be noted that this setting is deprecated since TYPO3 CMS 6.2 LTS and function `execute(TRUE)` should be used instead.

9.6. Example: Search for Keyword in Title

So now our initial task of finding a keyword in the title (independent from its position in the text) is able to be implemented with ease. First, we add the following method to the class `typo3conf/ext/simpleblog/Classes/Domain/Repository/BlogRepository.php`:

```
public function findSearchWord($search) {

   $query = $this->createQuery();

   $query->matching(
      $query->like('title','%'.$search.'%')
   );

   return $query->execute();
 }
```

For demonstration purposes however what if we wanted to implement the following:

- that the keyword should occur in the title and the description should be empty; or,
- that the title should read *TYPO3* exactly and the description should be *is fantastic*; or,
- that the title should at least match one of the words *Huey*, *Dewey*, or *Louie* (the list must be variable, because its source could be an external source).

```
public function findSearchWord($search, $words = array('Huey', 'Dewey', '

   $query = $this->createQuery();

   $query->matching(
      $query->logicalOr(
         $query->logicalAnd(
            $query->like('title', '%'.$search.'%'),
            $query->equals('description', '')
         ),
         $query->logicalAnd(
            $query->equals('title', 'TYPO3'),
            $query->like('description', '%is fantastic%')
          ),
         $query->in('title', $words)
      )
   );

   return $query->execute();
}
```

In addition, the result set should be limited to five records maximum and sorted alphabetically by title in ascending order:

```
...
```

```
   );

   $query->setOrderings(array('title' => \TYPO3\CMS\Extbase\Persistence\Quer

   $query->setLimit(5);

   return $query->execute();
```

The example above shows the limit hard-coded, which is definitely not worthwhile. It would be much better if the value could be set in TypoScript or in FlexForms, which we will explain in the next chapter.

9.7. Dynamic Search in Repository

We are already in the position to dynamically use the search functionality on our Blog listing page. The following new method in the Blog repository takes care of that:

```
public function findSearchForm($search) {

   $query = $this->createQuery();

   $query->matching(
      $query->like('title','%'.$search.'%')
   );

   $query->setOrderings(array('title' => \TYPO3\CMS\Extbase\Persistence\Q

   $query->setLimit(5);

   return $query->execute();

}
```

Now, a small search form in the list view of the Blog is required (`List.html`):

```
...
<h1>Blog List</h1>

    <f:form action="list" additionalAttributes="{role:'form'}">
        <div class="form-inline">
            <div class="form-group">
               <f:form.textfield name="search" value="{search}" class="form-
               <f:form.submit value="Search!" class="btn-xs btn-primary" />
            </div>
        </div>
    </f:form>

<br />

<ul class="list-group">
...
```

Attribute `value="{search}"` is a speciality; its purpose is to show the keyword as a pre-filled value again after submitting the form.

Finally, the Blog controller needs to be adjusted slightly:

```
/**
 * @param string $search
 */
public function listAction($search = '') {
   $this->view->assign('blogs', $this->blogRepository->findSearchForm($se
    $this->view->assign('search', $search);
}
```

We accept the POST request search as variable `$search` (don’t forget the annotation!). On first execution this variable will be empty.

Then we pass the keyword to the repository method `findSearchForm()` and additionally to the template in order to show the search term in the form.

Converting special characters

Extbase (respectively Fluid) takes care of converting special characters to HTML entities by applying the PHP function `htmlspecialchars()`. In order to get raw data, the ViewHelper `<f:format.raw>` can be used.

9.8. Side Note: Request Object

The solution so far seems to be elegant and practical but becomes problematic when we want to process multiple values. In this case, it would be more pragmatic to access the argument inside the method rather than in the method signature:

```
public function listAction() {
   if ($this->request->hasArgument('search')){
      $search = $this->request->getArgument('search');
   }

   $this->view->assign('blogs', $this->blogRepository->findSearchForm($se
   ...
```

The request (`$this->request`) features some more useful methods, which can be found in file `TYPO3\CMS\Extbase\Mvc\Request.php`:

`setDispatched($flag)`
: Sets the request status to `dispatched`.

`isDispatched()`
: Checks if the request has already been dispatched.

`setControllerObjectName($controllerObjectName)`
: Sets the controller object name (must be fully-qualified).

`getControllerObjectName()`
: Returns the object name of the currently active controller.

`setPluginName($pluginName = NULL)`
: Sets the plugin name.

`getPluginName()`
: Returns the plugin name.

`setControllerExtensionName($controllerExtensionName)`
: Sets the extension name.

`getControllerExtensionName()`
: Returns the extension name.

`getControllerExtensionKey()`
: Returns the extension key of the extension.

`setControllerActionName($actionName)`
: Sets the name of the action.

`getControllerActionName()`
: Returns the name of the action.

`setArgument($argumentName, $value)`
: Sets argument with the name `$argumentName` to value `$value`.

`getArgument($argumentName)`
: Returns argument with the name `$argumentName`.

`hasArgument($argumentName)`
: Checks if an argument with the name `$argumentName` exists.

`setArguments(array $arguments)`

Sets the entire arguments array.

`getArguments()`
: Returns all arguments.

`setFormat($format)`
: Sets the format, e.g. `html`, `xml`, `png`, `json`. The default is `html`.

`getFormat()`
: Returns the format.

If you are not working inside the controller, the request details must be determined via `$this->getControllerContext()->getRequest()` first.

Chapter 10. TypoScript and FlexForm Configuration

To configure extensions and plugins in TYPO3 CMS, TypoScript as well as FlexForms can be used.

The main difference between these two methods is scope – whereas the the granularity of TypoScript is limited to a page, a FlexForm can hold multiple configurations per page. This becomes important when an extension is used multiple times on a page with different configurations.

10.1. TypoScript

The configuration via TypoScript is usually carried out in file `setup.txt` in the directory `Configuration/TypoScript/` of the extension.

In order to use this file it needs to be included in the TypoScript template of the website (*Include from static*).

10.1.1. Setup Scope

First and foremost, there are three main keys:

```
plugin.tx_[lowercasedextensionname] {

}

module.tx_[lowercasedextensionname] {

}

config.extbase {

}
```

Where `[lowercasedextensionname]` must be replaced with the name of the extension in lowercase characters only, without any underscores – for example: `simpleblog`.

The meanings of the main keys as follows:

`plugin.tx_[lowercasedextensionname]`
: This configures a specific extension in the frontend (no separate configuration exists for the plugin).

`module.tx_[lowercasedextensionname]`
: This configures a specific extension in the backend.

`config.extbase`
: This configures Extbase and therefore all extensions (frontend and backend).

10.1.2. Sub-keys

Inside these main keys, several sub-keys exist – we will use `simpleblog` as the extension name:

```
plugin.tx_simpleblog {
   view {

   }
   persistence {

   }
   objects {

   }
   features {

   }
   mvc {

   }
   legacy {

   }
   settings {

   }
   _LOCAL_LANG {

   }
   _CSS_DEFAULT_STYLE {

   }
}
```

The meanings of the sub-keys as follows:

`view`
: This configures paths to templates, layouts and partials.

`persistence`
: This powerful key has wide-ranging control, for example IDs for reading and writing objects, dependency injection, single table inheritance, mapping and much more.

`objects`
: Replaces classes by stating the original and the new class. Extbase will redirect calls to the new class from then on. This key only exists inside of `config.tx_extbase` and `plugin.tx_[lowercasedextensionname]`.

`features`
: Some specific Extbase features can be enabled or disabled, such as setting the configuration to use the new Property Mapper (see Chapter 16) for example. This key only exists inside of `config.tx_extbase`.

`mvc`
: The request handler can be configured here. This key only exists inside of

`config.tx_extbase`.

`legacy`
: This allows the configuration of outdated settings. This key only exists inside of `config.tx_extbase`.

`settings`
: This key forms the user space, which can be used for arbitrary custom configuration.

`_LOCAL_LANG`
: Overwrites selective language labels of the extension.

`_DEFAULT_CSS_STYLE`
: Applies default CSS.

10.1.3. Option: `view`

Let’s have a closer look at the first sub-key `view`:

```
...
view {
   templateRootPaths = EXT:simpleblog/Resources/Private/Templates/
   partialRootPaths = {$plugin.tx_simpleblog.view.partialRootPath}
   layoutRootPaths = fileadmin/Layouts/
}
...
```

These sub-keys are self-explanatory and define the paths to directories for templates, partials and layouts. The interesting point is the way how they have been defined:

Path of the extension
: By using `EXT:`, followed by an extension key and a path, the directory in the extension’s directory is used.

Constants
: Constants can be defined in the file `Configuration/TypoScript/constants.txt` or in the constants field of every TypoScript template. There you can specify paths (as arrays only) and access those values in the setup. As an example, you could add a constant `path.template` and access its value by using `{$path.template}` in the setup.

Direct path
: The third example shows how a direct path can be named relatively from the DocumentRoot directory of the TYPO3 instance.

In TYPO3 CMS versions prior 6.2 LTS, these keys were named `templateRootPath`, `layoutRootPath` and `partialRootPath` (without trailing “s”). These keys are still working but are deprecated and will be removed in the future.

If you want to expand Fluid templates in Extbase, you usually have to copy all template files to a new directory, re-configure the paths (as explained above) and then edit the copied files. In cases where only one template file should be amended, this would create an unwanted redundancy.

In order to solve this issue, so-called “fallback paths” have been introduced in TYPO3 CMS 6.2 LTS: Fluid searches for the template file in the directory with the highest index, and – assuming the template file could not be found – takes the next folder, down to the lowest index.

These fallback logic works for templates, layouts as well as for partials: `templateRootPaths`, `partialRootPaths`, `layoutRootPaths`.

An example of a typical fallback usage is as follows:

```
plugin.tx_simpleblog {
   view {
      templateRootPath = EXT:simpleblog/Resources/Private/Templates/
   }
}

plugin.tx_simpleblog {
   view {
      templateRootPath >
      templateRootPaths {
         10 = fileadmin/simpleblog/templates
         20 = fileadmin/special/simpleblog/templates
      }
   }
}
```

`formatToPageTypeMapping`

This option holds a mapping that allows for the determination of the page type when the format changed.

```
plugin.tx_simpleblog {
   view.formatToPageTypeMapping {
      txt = 99
      pdf = 123
   }
}
```

Now it is possible to name the format in Fluid and Extbase takes care of the correct PageType:

```
<f:link.action arguments="{blog: blog}" format="txt">[plaintext]</f:link.act
```

10.1.4. Option: persistence

This sub-key compiles all configurations which have something to do with the persistence layer, typically the storage and classes.

```
...
persistence {
   storagePid = 17
   classes {
      Lobacher\Simpleblog\Domain\Model\Author {
         newRecordStoragePid = 27
         mapping {
            tableName = fe_users
            columns {
               name.mapOnProperty = fullname
            }
         }
      }
   }
}
...
```

`storagePid` (stdWrap)
: A comma-separated list of pages, which are available to Extbase for writing and reading. The first number represents the page UID for read and write operations (assuming it is not otherwise configured). All other pages are read-only. If this option is omitted, the `storagePid` is automatically `0` and therefore records are written to, and read from, the root page of the website.

```
persistence {
   storagePid.cObject = CONTENT
   storagePid.cObject {
   select {
      pidInList = 1,2
      recursive = 10
      selectFields = *
   }
   table = pages
   renderObj = TEXT
   renderObj {
      field = uid
      required = 1
      wrap = ,|
   }
   stdWrap.substring = 1
   }
}
```

This example shows how the property `recursive` can be simulated in TYPO3 CMS prior to 6.2 LTS.

`recursive`

The property `recursive` is available in TYPO3 since version 6.2 LTS. It specifies the maximum number of levels that should be taken into account. For example, if you would have a folder and under this folder a set of folders, the UID of the parent folder can be set in `storagePid` and with `recursive = 1` all sub-folders in the first level would be taken into account.

`enableAutomaticCacheClearing` (boolean)
Value `1` clears the cache of the current page whenever a record has been written or updated. This option is enabled by default, but is sometimes not required, e.g. if a record is shown on a different page to that where it is stored. In this case, the configuration `TCEMAIN.clearCacheCmd = UID` (where UID is the page where the records are displayed) can be used.

`updateReferenceIndex` (boolean)
If this option is enabled (value `1`) the reference index will be updated during a frontend process. This update is a complex task and has an impact on performance and is therefore disabled by default.

Option: persistence.classes

With this key it is possible to configure classes with their fully-qualified class name (e.g. `Lobacher\Simpleblog\Domain\Model\Author`) as sub-keys. Under these sub-keys, further sub-keys can be defined:

`newRecordStoragePid`
Defines a specific storage place for this class only (UID of the page or folder).

`mapping`
Configures the mapping of classes and tables inside Extbase with further sub-keys (see the following configuration options).

`mapping.tableName`
In order to use a different table than the default for the class, it can be specified here (e.g. table `fe_users` instead of `tx_simpleblog_domain_model_author`).

`mapping.recordType`
If a table is used as storage in multiple classes (the so-called *Single Table Inheritance*), this option is used to map the record set to the correct class. This is a unique identifier of a table and it is recommended to use the name of the class. In addition to this configuration, an appropriate TCA entry (`'type' => 'record_type'`) must exist.

`mapping.subclasses`
Subclasses are used inside a Single Table Inheritance (STI). For instance, subclasses `Car` and `Motorbike` could be derived from the superclass `Vehicle`. In this case, subclasses `Car` and `Motorbike` would be defined in class `Vehicle` by using this key. Inside the subclass configuration, `recordType` and `tableName` (of the superclass) are configured.

`mapping.columns`
This maps single columns. The name of the mapped column stands on the left side, followed by `.mapOnProperty =`, and the name of the column in the domain model on the right side, for example: `last_name.mapOnProperty = familyName`. If the only difference between the left and the right side is, that underscores are used on the left

but not on the right, and the first letter is in upper case, the mapping is not required, for example: `last_name.mapOnProperty = lastName`.

10.1.5. Option: objects

The purpose of this key is to overwrite classes. First, the name of the class which should be overwritten is stated, followed by the key/value pair `className` and the name of the replacement class.

```
# As soon as a backendInterface is required, Typo3DbBackend is loaded
config.tx_extbase {
   objects {
      TYPO3\CMS\Extbase\Persistence\Generic\Storage\BackendInterface {
         className = TYPO3\CMS\Extbase\Persistence\Storage\Typo3DbBackend
   }
}
# In order to load your own Persistence Backend, the configuration
# can be overwritten
config.tx_extbase {
    objects {
        TYPO3\CMS\Extbase\Persistence\Generic\Storage\BackendInterface {
            className = Vendor\ExtensionName\Persistence\Backend
        }
    }
}
```

This allows you to add a standard namespace to all Fluid templates. Enter the following in your TypoScript setup:

```
config.tx_extbase {
   objects {
      TYPO3\CMS\Fluid\Core\Parser\TemplateParser {
         className = Lobacher\Simpleblog\View\TemplateParser
      }
   }
}
```

And the following to the file `Lobacher\Simpleblog\View\TemplateParser.php`:

```
<?php
namespace Lobacher\Simpleblog\View;
class TemplateParser extends TYPO3\CMS\Fluid\Core\Parser\TemplateParser {
   protected $namespacesBase = array();
   public function initializeObject() {
      $this->namespacesBase = $this->namespaces += array(
         'simpleblog' => 'Lobacher\Simpleblog\ViewHelpers'
      );
   }

   protected function reset() {
      $this->namespaces = $this->namespacesBase;
   }
}
?>
```

This makes the need of using the namespace statement in all templates obsolete, e. g. `{namespace simpleblog = Lobacher\Simpleblog\ViewHelpers}`. ViewHelpers can be addressed directly now (`<simpleblog:viewHelper>`) without declaring it as a namespace in the template.

10.1.6. Option: features

The `features` option enables developers and integrators to turn Extbase features on and off. At this point in time the following configurations are available:

`rewrittenPropertyMapper`
: Since TYPO3 CMS version 6.1, Extbase uses the new Property Mapper[30] (which has been ported from TYPO3 Flow) by default. It has been available since version 4.7 but not activated (value: `1`) by default until version 6.1.

`skipDefaultArguments`
: If this value is set to `1` (default is `0`), the default controller and/or the default action is not taken into account when URLs are generated by the URI builder.

`ignoreAllEnableFieldsInBe`
: If this value is set to `1` (default is `0`), all *Enable Fields* (such as *hidden*, *deleted*, etc.) are ignored in the backend context.

10.1.7. Option: mvc

You can configure the three Extbase request handlers (frontend, backend and CLI) with this option and therefore overwrite the default settings where required. The default is:

```
config.tx_extbase {
   mvc {
      requestHandlers {
         TYPO3\CMS\Extbase\Mvc\Web\FrontendRequestHandler = TYPO3\CMS\Extbas
         TYPO3\CMS\Extbase\Mvc\Web\BackendRequestHandler = TYPO3\CMS\Extbase
         TYPO3\CMS\Extbase\Mvc\Cli\RequestHandler = TYPO3\CMS\Extbase\Mvc\Cl
      }
   }
}
```

10.1.8. Option: legacy

Out-dated and obsolete options can be configured with this option. Currently only the following key exists:

`enableLegacyFlashMessageHandling`

In older versions FlashMessages can only be stored in a container. Since version 6.1, TYPO3 CMS also allows for the addition of FlashMessages to a queue in a plugin. In order to activate this the old FlashMessageHandling must be disabled:

```
config.tx_extbase.legacy.enableLegacyFlashMessageHandling = 0
```

The default value is `1`.

10.1.9. Option: settings

The key `settings` allows developers to define arbitrary key/value pairs which can be accessed in the controller as well as in templates.

```
plugin.tx_simpleblog {
   settings {
      blog {
         max = 5
      }
   }
}
```

To access the value in the controller the variable `$this->settings['blog']['max']` can be used. In order to access the value in a template `{settings.blog.max}` is used.

10.1.10. Option: _LOCAL_LANG

Language labels of extensions can be overwritten by using this option. The key `default` stands for the default language otherwise the two-letter language code (ISO code) is used.

```
plugin.tx_simpleblog {
   _LOCAL_LANG {
      default {
         linkLabel = English link
      }
      de {
         linkLabel = Deutscher Link
      }
   }
}
```

10.1.11. Option: _CSS_DEFAULT_STYLE

In order to set custom CSS styles the `_CSS_DEFAULT_STYLE` option can be used. For example, it is possible to colour an input field red if the data entered is invalid:

```
plugin.tx_simpleblog._CSS_DEFAULT_STYLE (
   .f3-form-error {
      background-color:#FF9F9F;
      border: 1px #FF0000 solid;
   }
)
```

10.2. FlexForms

By using TypoScript a specific configuration can be applied on a page-by-page basis. However if you require multiple configurations – for example because more than one extension has been placed on a page – the TypoScript approach reaches its limits quickly.

A better solution is to store the configuration in the content element (the plugin record itself), e.g. in a new column of the database table which is responsible for the content element. To avoid creating multiple columns for every plugin, which are not used by any other plugins, the TYPO3 project has agreed on a specific field: `pi_flexform` in the database table `tt_content`. A specific XML structure (the so-called "FlexForm") is stored in this field, which in turn provides additional form fields in the backend.

The basis of the FlexForm is a XML file with an unusual syntax. In order to use this file a few commands must be added to the file `ext_tables.php` as shown below:

```
$pluginSignature = 'simpleblog_bloglisting';
$TCA['tt_content']['types']['list']['subtypes_addlist'][$pluginSignature] =
\TYPO3\CMS\Core\Utility\ExtensionManagementUtility::addPiFlexFormValue( $plu
```

These instructions load the file `flexform_bloglisting.xml` in the `directory` `typo3conf/ext/simpleblog/Configuration/FlexForms` which we have to create first:

```
cd [DocumentRoot]
mkdir typo3conf/ext/simpleblog/Configuration/FlexForms
cd typo3conf/ext/simpleblog/Configuration/FlexForms
```

Now create the file `flexform_bloglisting.xml` with the following content:

```
<T3DataStructure>
   <sheets>
      <sDEF>
         <ROOT>
            <TCEforms>
               <sheetTitle>Blog Config</sheetTitle>
            </TCEforms>
            <type>array</type>
            <el>

            </el>
         </ROOT>
      </sDEF>
   </sheets>
</T3DataStructure>
```

This is the foundation for further settings, which are placed between the `<el>` and `</el>` tags. Please note that we can show only a simple example in this book, because FlexForms are very powerful and complex and you could write a whole other book about this topic. Additional documentation is of course available.[31] You find further field types in Section A.2.

10.2.1. FlexForm Configuration

This chapter describes how to implement a configuration option by using a FlexForm, which allows editors to define the number of Blogs shown in the list view.

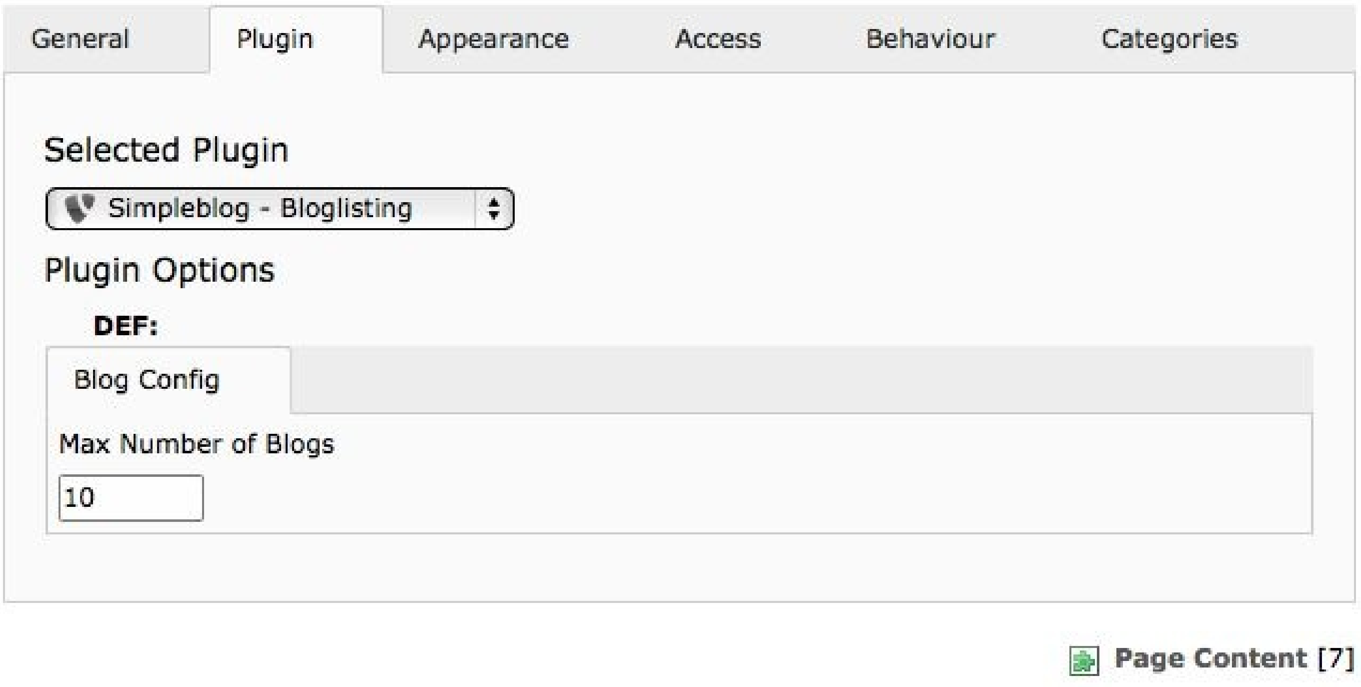
The following code inside the `<el>` and `</el>` tags take care of that:

```
<settings.blog.max>
   <TCEforms>
      <label>Max Number of Blogs</label>
      <config>
         <type>input</type>
         <size>2</size>
         <eval>int</eval>
         <default>10</default>
      </config>
   </TCEforms>
</settings.blog.max>
```

It is important, that every element is wrapped by a tag with the name starting with `settings` and a dot. This tells Extbase that the configuration should be read automatically. The other fields define a TCEforms-field of type `input` with a length of 2 characters and a pre-filled default value of `10`.

After clearing the cache in the TYPO3 backend, the panel *Plug-In* of the content element shows the new configuration option:

Figure 10.1. Content element configuration via FlexForm



Selected Plugin

Simpleblog - Bloglisting

Plugin Options

DEF:

Max Number of Blogs

10

Accessing the value works the same as for the TypoScript configuration described earlier; use `$this->settings['blog']['max']` in the controller and `{settings.blog.max}` in the Fluid view.

Bug in the configuration determination

If the same configuration variable is used in the FlexForm and in TypoScript the value from the FlexForm has a higher priority and is used. This is intentional. However if you delete the value from the FlexForm, you would expect the value from TypoScript is then used. This is not the case due to a bug[32] in Extbase. In this case, Extbase sets the value to `0` and always uses this FlexForm value.

A possible work-around is a Helper function, which can be seen in action in method `injectConfigurationManager` in the “News” extension by Georg Ringer.[33]

10.2.2. Switchable Controller Actions (SCA)

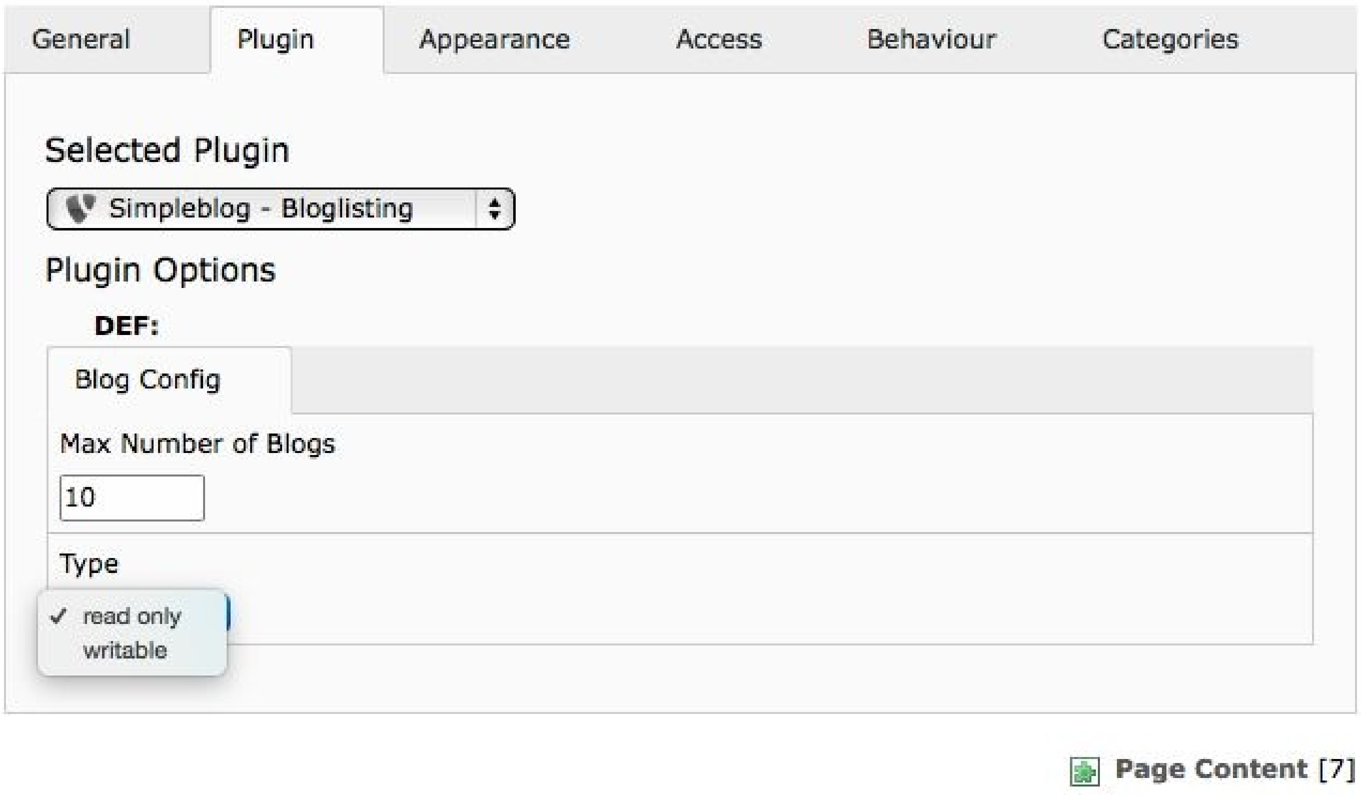
Switchable Controller Actions (SCA) let you restrict access to specific controller/action combinations per plugin (in the FlexForm configuration). This means the plugin can be configured in a way to restrict the execution of only those controllers and actions which have been specified in the FlexForm.

```
<switchableControllerActions>
   <TCEforms>
      <label>Type</label>
      <config>
         <type>select</type>
         <items type="array">
            <numIndex index="0" type="array">
               <numIndex index="0">read only</numIndex>
               <numIndex index="1">Blog->list;Blog->show</num
            </numIndex>
            <numIndex index="1" type="array">
               <numIndex index="0">writable</numIndex>
               <numIndex index="1">Blog->list;Blog->show;Blog
            </numIndex>
         </items>
      </config>
   </TCEforms>
</switchableControllerActions>
```

By placing this code between the `<el>` and `</el>` tags a dropdown box appears in the backend. Selecting option 1 (“read only”) means the plugin can only execute the combination `Blog->list` and `Blog->show`. Option 2 (“writable”) also enables `Blog->deleteConfirm` and `Blog->delete`. Additionally, further actions such as `update` can be added.

SCA becomes active as soon as the plugin configuration is accessed in the backend, an option has been selected and the new configuration saved. These steps store the FlexForm configuration to the database.

Figure 10.2. Configuration of Switchable Controller Actions



Selected Plugin

Simpleblog - Bloglisting

Plugin Options

DEF:

Blog Config

Max Number of Blogs

10

Type

✓ read only
writable

For the steps described in the following sections of this book it is required to remove the FlexForm again.

Persevering FlexForms

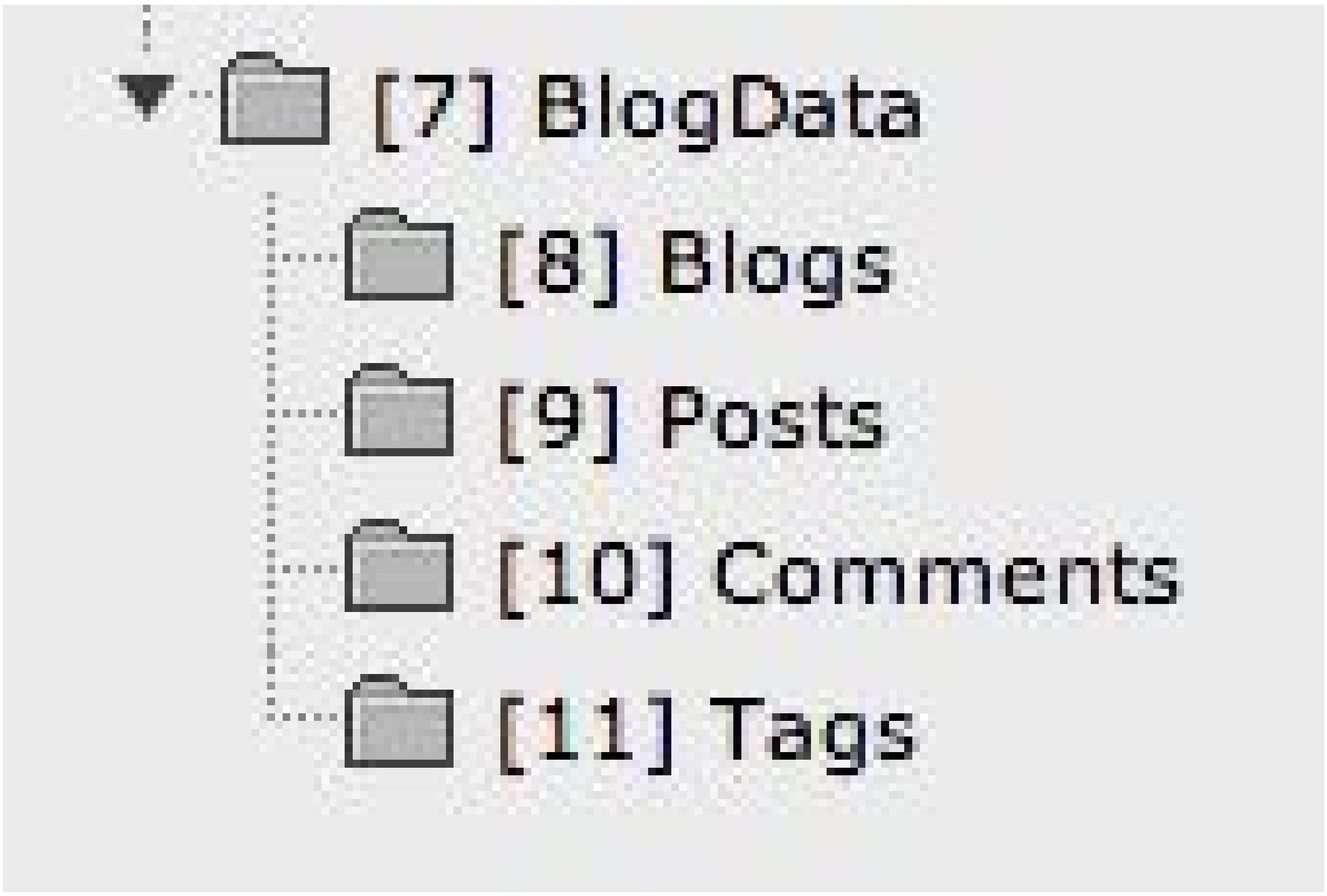
Due to the way in which FlexForms are managed internally it might be difficulty to get rid of them. Basically there are two options:

- delete the plugin, delete the three code lines from the file `ext_tables.php`, clear the cache and re-implemented the plugin; or,
- empty the field `pi_flexform` of the content element in the database directly and remove the lines from file `ext_tables.php`.

10.3. TypoScript for the Next Sections of this Book

Create the following five folders in the TYPO3 backend (see Figure 10.3) in preparation for the next steps. You will most likely get different UIDs than the values shown here. Note your values and use them in the TypoScript below.

Figure 10.3. Folders used for Blog data



Then let us edit the TypoScript in the file `typo3conf/ext/simpleblog/Configuration/TypoScript/setup.txt`:

```
plugin.tx_simpleblog {
   view {
      templateRootPath = {$plugin.tx_simpleblog.view.templateRootPath}
      partialRootPath = {$plugin.tx_simpleblog.view.partialRootPath}
      layoutRootPath = {$plugin.tx_simpleblog.view.layoutRootPath}
   }
   persistence {
      storagePid = 7,8
      recursive = 1
      classes {
         Lobacher\Simpleblog\Domain\Model\Blog {
            newRecordStoragePid = 8
         }
         Lobacher\Simpleblog\Domain\Model\Post {
            newRecordStoragePid = 9
         }
         Lobacher\Simpleblog\Domain\Model\Comment {
            newRecordStoragePid = 10
         }
         Lobacher\Simpleblog\Domain\Model\Tag {
            newRecordStoragePid = 11
         }
      }
   }
   settings {
      blog {
         max = 5
      }
   }
}
```

We have implemented our custom configuration by using the keyword `settings`. In order limit the listing to only five Blog records, we have to retrieve this information in the `listAction()` in file `typo3conf/ext/simpleblog/Classes/Controller/BlogController.php`. The value can be accessed by reading `$this->settings[]`. The limit is then passed to the repository function, which reads the Blogs from the database.

```
public function listAction() {
   if ($this->request->hasArgument('search')){
      $search = $this->request->getArgument('search');
   }

   $limit = ($this->settings['blog']['max']) ?: NULL;

   $this->view->assign('blogs', $this->blogRepository->findSearchForm($searc
   $this->view->assign('search', $search);
}
```

We now open the file `typo3conf/ext/simpleblog/Classes/Domain/Repository/BlogRepository.php` and edit the repository function `findSearchForm` as follows:

```
/**
 * @param string $search
 * @param int $limit
 * @return array|\TYPO3\CMS\Extbase\Persistence\QueryResultInterface
 */
public function findSearchForm($search,$limit) {

   $query = $this->createQuery();

   $query->matching(
      $query->like('title','%'.$search.'%')
   );

   $query->setOrderings(array('title' => \TYPO3\CMS\Extbase\Persistence\Quer

   $limit = (int)$limit;
   if ($limit > 0) {
      $query->setLimit($limit);
   }

   return $query->execute();
}
```

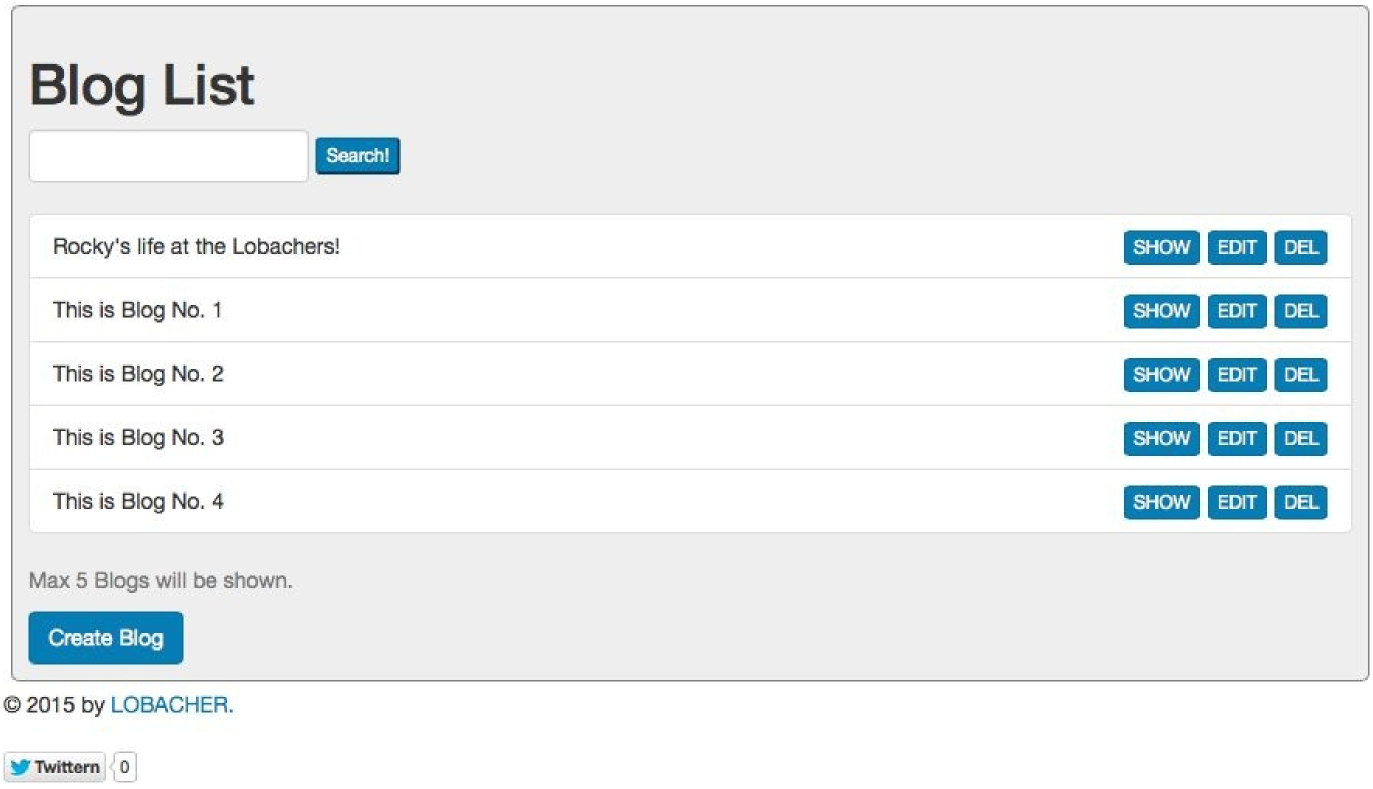
Firstly, `$limit` has to be added as a method parameter. Unfortunately the `@param` annotation is insufficient to convert `$limit` into an integer in this case. As a consequence, we have to cast the value by using `(int)`.

If `$limit` is greater than `0` we set the limit for the query.

We can also access the value in the frontend by using `{settings.blog.max}`. Open the file `typo3conf/ext/simpleblog/Resources/Templates/Blog/List.html` and add the following below the `<ul>` list:

```
...
</ul>

<p class="text-muted">Max {settings.blog.max} Blogs will be shown.</p>

<f:link.action action="addForm" class="btn btn-primary">Create Blog</f:link.
...
```

Figure 10.4. Output of the limit as configured



Accessing settings in other areas

Accessing TypoScript and FlexForm settings is only possible in the controller and in the view. This is by design as logically this is where these functions belong.

However under certain circumstances it might be useful to access settings in the model or repository. This is also possible but requires some extra effort:

```
$objectManager = \TYPO3\CMS\Core\Utility\GeneralUtility::makeInstance(
$configurationManager = $objectManager->get( 'TYPO3\\CMS\\Extbase\\Cor
$settings = $configurationManager->getConfiguration( \TYPO3\CMS\Extbas
```

Variable `$settings` now contains the TypoScript/FlexForm data.

10.4. TypoScript for Backend Modules

It is also possible in Extbase to use TypoScript for backend modules. The problem however, is that no pages exist in which the TypoScript code can be stored (the backend does not require pages as such).

We can get around this by using the static TypoScript for our configuration. As before, the file `typo3conf/ext/simpleblog/Configuration/TypoScript/setup.txt` can be used for this purpose. The key `module.tx_simpleblog` has been designed to achieve this. Should this key already hold some settings you should copy them before proceeding.

```
...
module.tx_simpleblog < plugin.tx_simpleblog…
```

---

[30] http://forge.typo3.org/projects/typo3v4-mvc/wiki/PropertyMapper_rework

[31] http://wiki.typo3.org/Extension_Development,_using_Flexforms and http://docs.typo3.org/typo3cms/CoreApiReference/DataFormats/T3datastructure/Index.html

[32] http://forge.typo3.org/issues/51935

[33] https://git.typo3.org/TYPO3v4/Extensions/news.git/blob/dc0a5ad9a3ff77dffd1a5cbabdf587a28d244115:/Classes/Control

Chapter 11. Validation and Error Handling

In order to process data – which the system expects in a correct format – the data must be checked and validated. Extbase provides a comprehensive validation framework for this task.

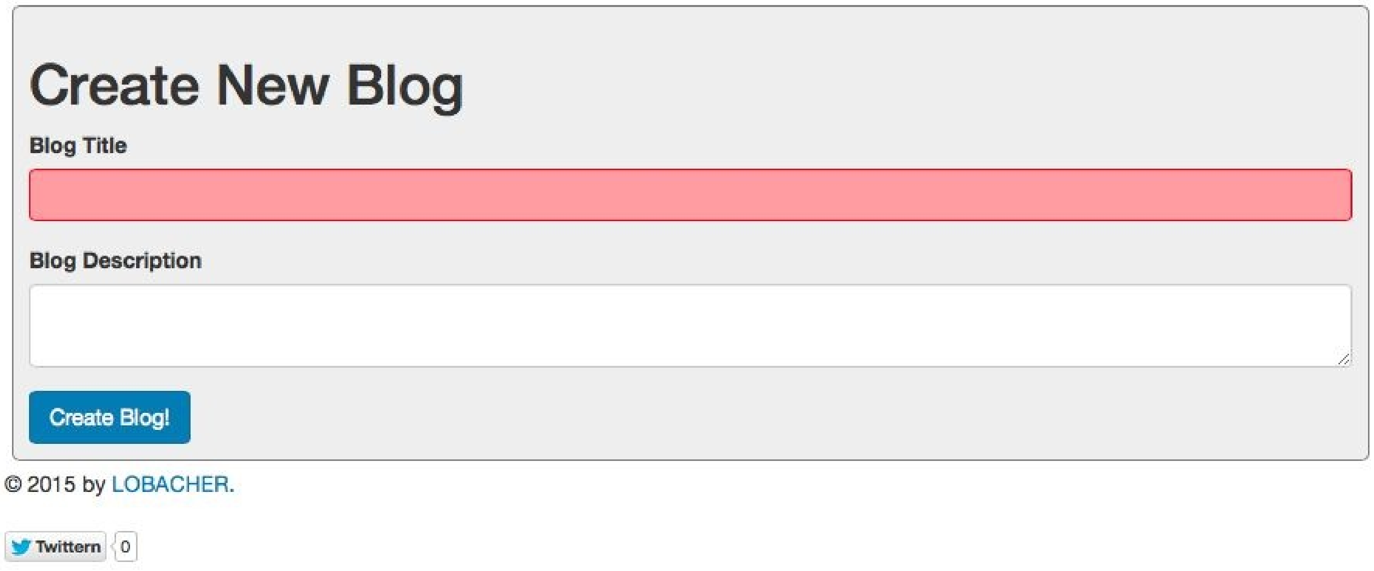
11.1. Error Handling

Before we take a look at the data validation in detail, we should explore error handling, which takes care of detecting invalid objects and lets us treat them as errors.

If you go to the Blog list, click *Create Blog!* and leave the Blog title empty. You will notice that the Blog title input field is coloured red. The HTML source code shows the CSS class `f3.form-error`, which has been added by Extbase. The previously-included TypoScript introduces this specific style by the section `_CSS_DEFAULT_STYLE`.

```
<input class="form-control f3-form-error" type="text" name="tx_simpleblog_bl
```

Figure 11.1. Highlighting of an invalid field



For the purpose of this example however, we would like to achieve not only an additional style but also an error message. This requires a little bit more work. As soon as a validation error occurs, some specific error objects are created which can be displayed via Fluid. The following section shows how to implement a partial that outputs all errors. You can customise this partial to meet your specific requirements later.

First, create a new file `Error.html` with the following content in folder `typo3conf/ext/simpleblog/Resources/Partials/`:

```
<f:form.validationResults for="{object}">

    <f:for each="{validationResults.flattenedErrors}" as="errors" key="prope
        {propertyPath}
        <ul>
            <f:for each="{errors}" as="error">
                <li>
                    {error} | {error.code} |
                    Arguments: <f:for each="{error.arguments}" as="argument"
                </li>
            </f:for>
        </ul>
    </f:for>

</f:form.validationResults>
```

The ViewHelper `f:form.validationResults` is responsible for processing the error object. The option `for` specifies the object for which the output should appear. We should keep this flexible and use the value `{object}`. This means we have to pass some information to the partial when we include it.

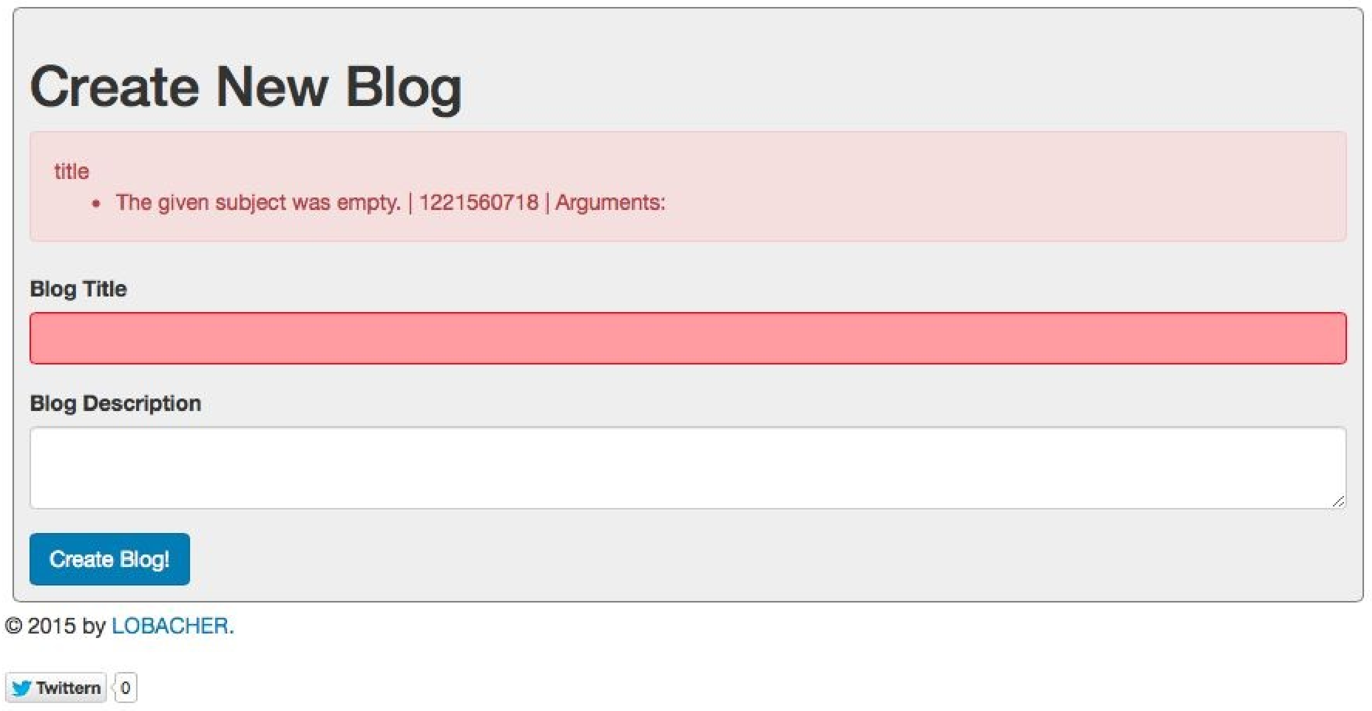
We can cycle through all errors flagged by applying the `for`-ViewHelper. The `propertyPath` is the path to the property of the domain object which was invalid and threw the error. Every invalid property could have multiple errors of course and those are addressed by the next iteration.

The error itself is stored in `error` while the error code is stored in `error.code` and possible arguments stored in `error.arguments`.

Finally, we need to include the error partial in the form partial in file `typo3conf/ext/simpleblog/Resources/Partials/Blog/Form.html`:

```
<h1>{headline}</h1>
<div class="alert alert-danger"><f:render partial="Error" arguments="{object
...
```

Figure 11.2. Output an error



Create New Blog

title

- The given subject was empty. | 1221560718 | Arguments:

Blog Title

Blog Description

We will leave the error handling as it stands now for the time being. Later, we will make the error message multi-lingual and output it at the right spot. Let’s have a look at the validation functionality first.

11.2. Validation Overview

Generally speaking in domain-driven design, validation is part of the model because it reflects a part of the business logic. There are three occurrences where validation plays an important role:

Property Validation
: Validates a property of a model.

Object Validation
: Validates an entire object at a time.

Action Validation
: Validates the input parameter of an action as soon as the action is executed.

When does a validation happen?

It is also important at which position a validation occurs inside an Extbase process – namely always when the action is entered. Whereas TYPO3 Flow also validates when entering the repository, Extbase has this single window for a validation. In order to prevent a validation from occuring, the following annotation can be set at the action:

```
/**
 * @ignorevalidation $blog
 */
public function addAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog)
...
```

At the time of the old Property Mapper, this annotation was called `@dontvalidate`.

As a matter of principle, the action `errorAction()` is called when an error occurs. This action exists in the action controller `TYPO3\CMS\Extbase\Mvc\Controller\ActionController.php` but can be overwritten with your own method `errorAction()`.

11.3. Property Validation

As the name indicates, the property validation is executed directly at the properties of the domain model.

Open file `typo3conf/ext/simpleblog/Classes/Domain/Model/Blog.php` and have a look at property `$title`:

```
/**
 * This is the title of the blog
 * @var \string
 * @validate NotEmpty
 */
 protected $title;
```

Due to the fact that we marked this property as `required` earlier the Extension Builder takes care of two things: it adds the annotation `@validate NotEmpty` and extends the Table Configuration Array (TCA) by adding the section `'eval' => 'trim,required'`. The latter we will cover later, but let’s have a closer look at the annotation which starts with `@validate` followed by so-called validators.

11.3.1. Built-in Validators

Currently, Extbase supports 13 built-in validators, which can be used as annotations:

`Alphanumeric`
: Checks if the value consists of characters a-z, A-Z, 0-9 only.

`Boolean(is=TRUE|FALSE)`
: Checks if the value is either `TRUE` or `FALSE`, thus `Boolean(is=TRUE)` or `Boolean(is=FALSE)`.

`Conjunction` / `Disjunction`
: Checks if *all* given validators are `TRUE` (`Conjunction`) or at least *one* validator (`Disjunction`), for example: `Conjunction(0=NoEmpty,1=Boolean(is=TRUE),2=...)`

`DateTime`
: Checks if the value is a valid DateTime object.

`EmailAddress`
: Checks if the value is a valid email address.

`Float`
: Checks if the value is a valid floating point number.

`Integer`
: Checks if the value is a valid integer number.

`NotEmpty`
: Checks if the value is not empty (`NULL` or an empty string).

`NumberRange(startRange,endRange)`
: Checks if the value is between `startRange` and `endRange`.

`Number`
: Checks if the value is a valid number.

`RegularExpression(regularExpression)`
: Checks if the value matches the given regular expression.

`StringLength(minimum,maximum)`
: Checks if the value is a string and its length is between `minimum` and `maximum`.

`String`
: Checks if the value is a string.

`Text`
: Checks if the value is a valid text and, therefore, does not contain any XML tags.

11.3.2. Multiple Validators

Validators can be combined by comma-separating them or by writing each validator in one line:

```
/**
 * This is the title of the blog
 * @var \string
 * @validate NotEmpty, StringLength(minimum=5,maximum=10), Text
 */
 protected $title;
```

or:

```
 /**
 * This is the title of the blog
 * @var \string
 * @validate NotEmpty
 * @validate StringLength(minimum=5,maximum=10)
 * @validate Text
 */
 protected $title;
```

Assuming all validators result in `TRUE`, the property is considered as validated and therefore “valid”. If only one validator fails and returns `FALSE`, the property is “invalid”. It is important to note that all validators are checked, even if the first one already returns `FALSE`.

11.3.3. Custom Validators

Despite the great selection of 13 existing validators, custom validators are sometimes required. They can be implemented quickly and added to the existing annotations easily.

The following section describes how to develop a validator which counts the number of words of a property. At a configurable threshold, an error message is shown. This prevents the length of the Blog title becoming too long.

The new validator is used in the model class `typo3conf/ext/simpleblog/Classes/Domain/Model/Blog.php`:

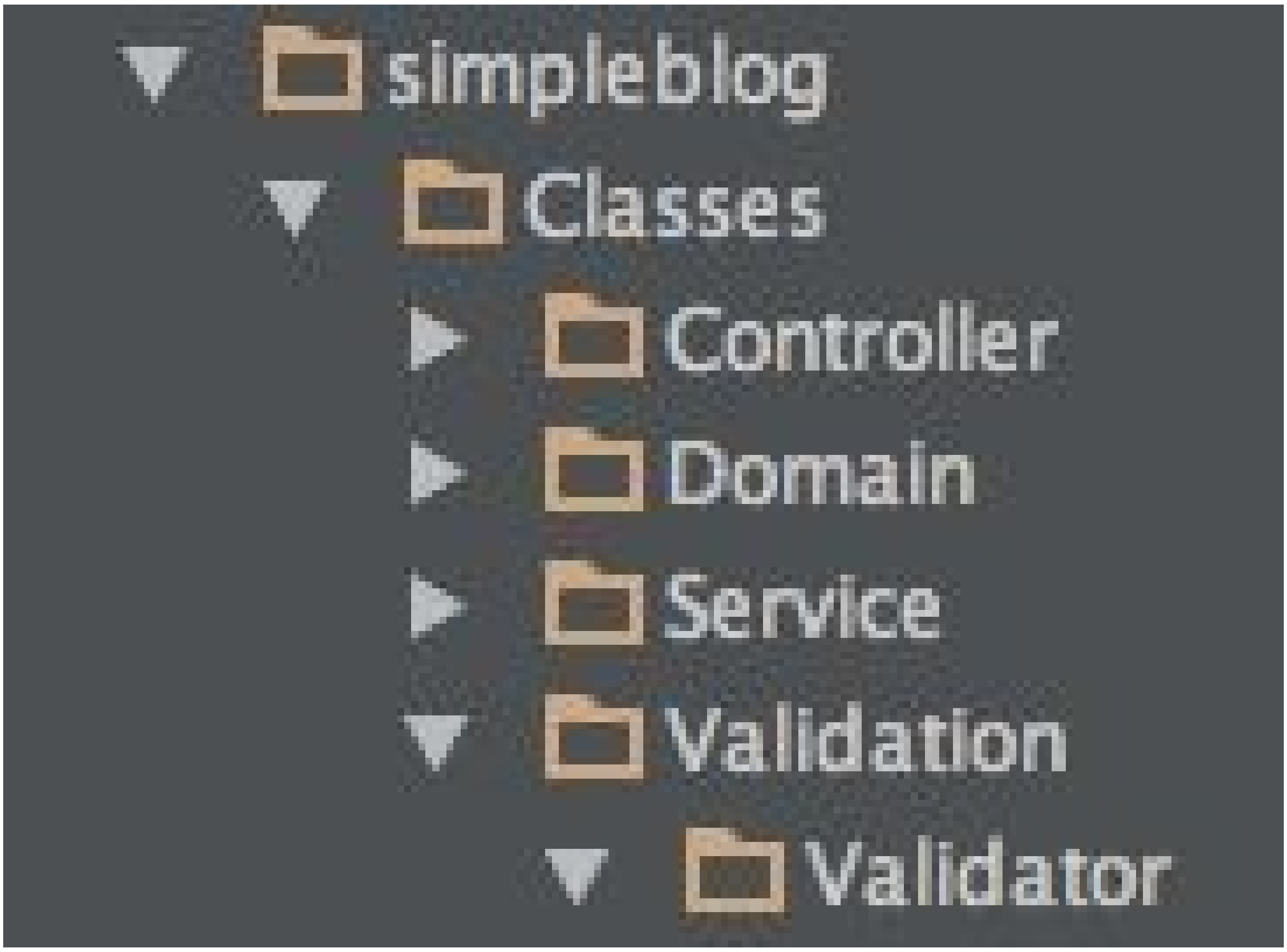
```
/**
 * This is the title of the blog
 * @var \string
 * @validate NotEmpty, \Lobacher\Simpleblog\Validation\Validator\WordValidat
 */
 protected $title;
```

Alternatively, an abbreviated form can be used by writing vendor, extension and validator name in the form `Vendorname.Extensionname:Validatorname`:

```
/**
 * This is the title of the blog
 * @var \string
 * @validate NotEmpty, Lobacher.Simpleblog:Word(max=3)
 */
 protected $title;
```

An important requirement is that the file `[Validatorname]Validator.php` (e.g. `NotEmptyValidator.php`) is stored in the folder `typo3conf/ext/simpleblog/Classes/Validation/Validator/`. Therefore we create a new directory `Validation` inside `Classes` and a subdirectory `Validator`.

Figure 11.3. Directory structure for custom validators



A new file with the following content is stored in this directory:

```
<?php
namespace Lobacher\Simpleblog\Validation\Validator;

class WordValidator extends \TYPO3\CMS\Extbase\Validation\Validator\Abstract

   protected $supportedOptions = array(
        'max' => array(PHP_INT_MAX, 'The maximum word count to accept', 'int
    );

   public function isValid($property) {

        $max = $this->options['max'];

        if (str_word_count($property, 0) <= $max) {
         return TRUE;
      } else {
         $this->addError('Reduce the amount of words - max '.$max.' are allo
         return FALSE;
      }
   }

}
```

```
?>
```

The validator class must be derived from the abstract validator `\TYPO3\CMS\Extbase\Validation\Validator\AbstractValidator`. The only function we have to implement is `isValid()`. If this method returns `TRUE`, the property is valid, if it returns `FALSE`, the property is invalid. The property itself is passed as a method parameter and is available inside the function.

Additionally you have to add a protected member `$supportedOptions` that defines the allowed parameters of the validator (`max` in our example).

This member has to be an associative array, with the option names as keys, and the option configuration as values. The option configurations are numerically indexed arrays, where the array entries have the following meanings:

Index 0
: default value for the option (mixed)

Index 1
: description of the option (string)

Index 2
: type of the option (string, values are “string”, “integer” etc.)

Index 3
: whether the option is required (boolean, optional).

An example from the NumberRangeValidator, not using the fourth option:

```
protected $supportedOptions = array(
    'minimum' => array(0, 'The minimum value to accept', 'integer'),
    'maximum' => array(PHP_INT_MAX, 'The maximum value to accept', 'integer'
    'startRange' => array(0, 'The minimum value to accept', 'integer'),
    'endRange' => array(PHP_INT_MAX, 'The maximum value to accept', 'integer
);
```

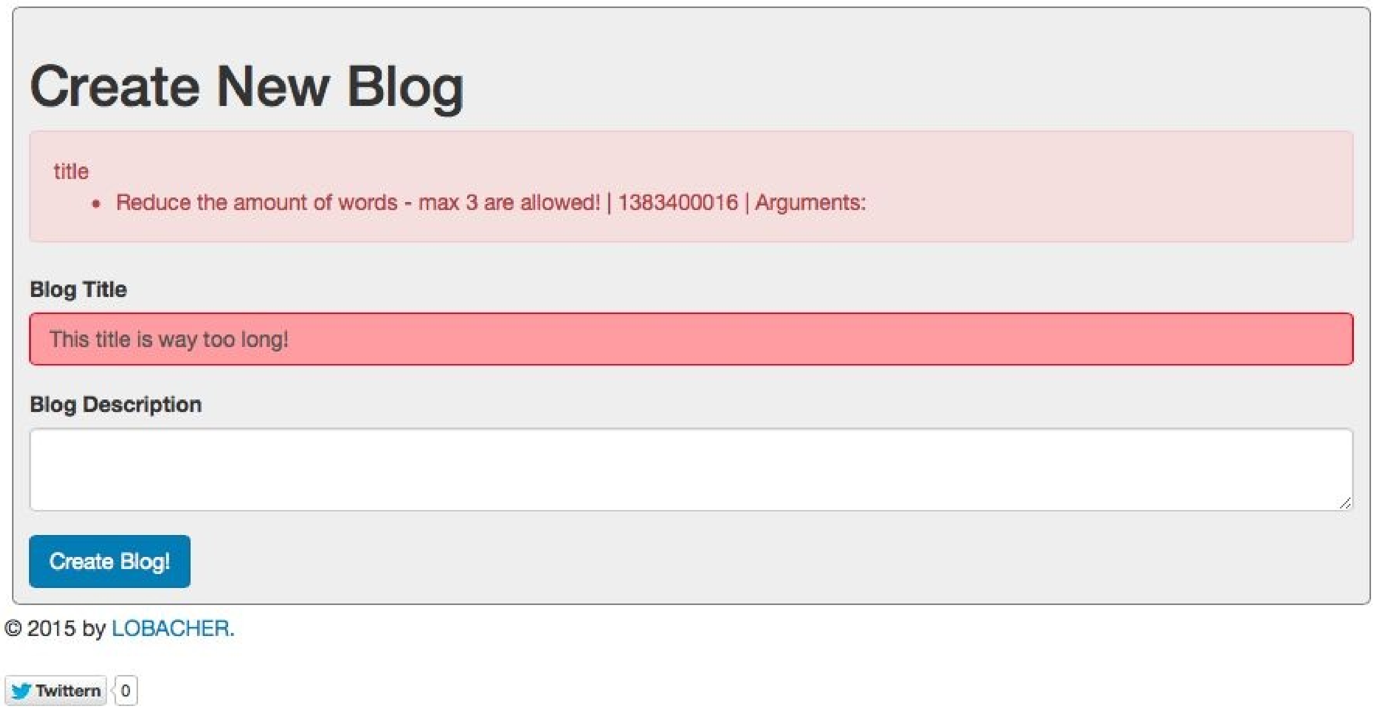
Return value of `isValid()`

It is important to understand that the behaviour of the `isValid()` function has been modified so that the return value does not indicate if the validation was successful or not. The relevant part is in fact the `addError()` call. If you return `FALSE` only, without adding the error, the validator is deemed as valid.

All of the options which have been stated in the annotation can be accessed via `$this->options`. We use the threshold `max` to define the maximum number of words allowed.

In the `FALSE` case an error should be shown. This is achieved by the `addError()` call. The first parameter is the error message (we will extract the text and store it in a language file later). The second parameter is a unique error number (it is recommended to use the current timestamp, because this eliminates any ambiguity created by country-specific date formats).

Figure 11.4. Error message generated by custom validator


Create New Blog

title

- Reduce the amount of words - max 3 are allowed! | 1383400016 | Arguments:

Blog Title

This title is way too long!

Blog Description

© 2015 by LOBACHER.

11.4. Object Validation

Where the property validator allowed us to validate individual properties one at a time, occasionally it is required that we validate an object as a whole, typically in order to validate two or more properties if they are somehow connected. This could be a password and its verification field (repeat password) or a checkbox “call me back” and the check to ensure the second field contains a valid phone number. This is the purpose of object validation.

For demonstration purposes we will implement a method that checks if someone entered “Joomla” as the Blog title and “is great” as the description – so we can prevent this from happening. Joomla developers can feel free to use “Drupal”, “Wordpress” or “42” instead.

The object validator must be named exactly the same as the object (in our example: `BlogValidator`) and stored in the directory `Lobacher\Simpleblog\Domain\Validator` named `BlogValidator.php`:

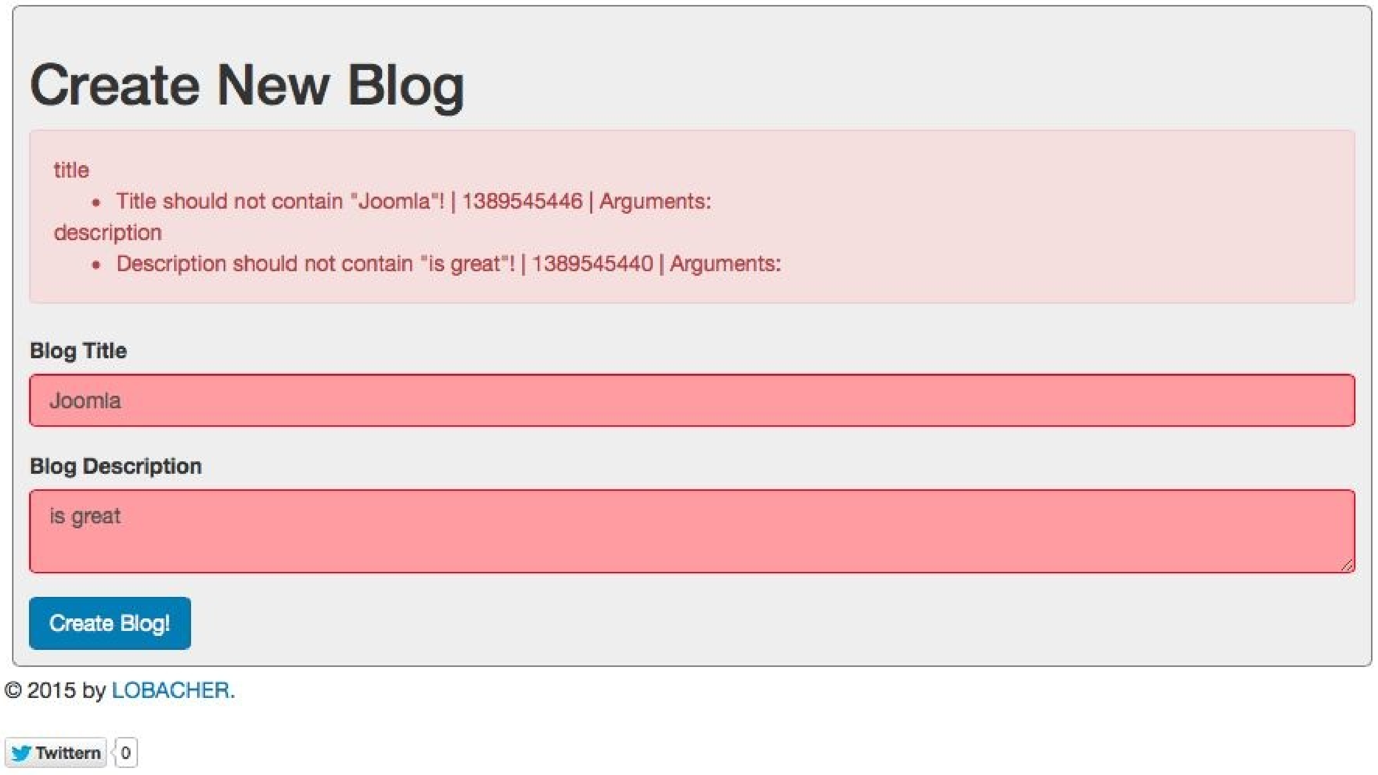
```
<?php
namespace Lobacher\Simpleblog\Domain\Validator;

class BlogValidator extends \TYPO3\CMS\Extbase\Validation\Validator\Abstract

   /**
    * Validates the given value
    *
    * @param mixed $object
    * @return bool
    */
   protected function isValid($object) {

        if (    preg_match('/Joomla/i',$object->getTitle()) &&
                preg_match('/is great/i',$object->getDescription())) {

            $this->result->forProperty('title')
                ->addError(
                    new \TYPO3\CMS\Extbase\Error\Error(
                        'Title should not contain "Joomla"!', 1389545446));

            $this->result->forProperty('description')
                ->addError(
                    new \TYPO3\CMS\Extbase\Error\Error(
                        'Description should not contain "is great"!', 138954
            return FALSE;
        } else {
            return TRUE;
        }
    }
}
```

```
?>
```

Similarly to the property validator there is also a method `isValid()`, which must return `TRUE` or `FALSE`.

The difference though is that the method parameter includes the whole object (the Blog) and therefore we have to use the getters to access the properties. Additionally, the error statement is a little bit more complicated because it is necessary to define which property threw validation errors (`$this->result->forProperty(...)`) as well as “attaching” one or more error objects to them which is performed by `->addError(...)`.

Figure 11.5. Error message generated by object validator



Create New Blog

title
- Title should not contain "Joomla"! | 1389545446 | Arguments:

description
- Description should not contain "is great"! | 1389545440 | Arguments:

Blog Title

Joomla

Blog Description

is great

Bug in Validator Resolver

Beginning with TYPO3 CMS 6.2.0 there is a bug in the Validator Resolver which is propably not solved till today. The bug results in displaying the error messages twice with the method above.

So you could either patch your installation with the file found here: `https://forge.typo3.org/projects/typo3cms-core/repository/revisions/ae317f2ae12c7e3743b5b5810c89c7223c9a902c` or you can comment out the following lines in the file `typo3/sysext/extbase/Classes/Validation/ValidatorResolver.php` (aprox. line 156:

```
                  // @todo: remove check for old underscore model name syntax
                              if (strpbrk($methodParameter['type'], '_\\') ==
                                     $typeValidator = $this->createValidator
```

```
//                      } elseif (preg_match('/[\\_]Model[\\_]/', $meth
//                          $possibleValidatorClassName = str_replace(a
//                              $typeValidator = $this->createValidator
                        } else {
                                $typeValidator = NULL;
```

A different directory for object validators

According to new conventions, all validators should be stored in the directory `Classes/Validation/Validator/`. However it seems that Extbase has a bug, because the object validator requires a validator in the directory `Classes/Domain/Validator`.

Method `TYPO3\CMS\Core\Utility::translateModelNameToValidatorName()` seems to be responsible for this because it replaces `Domain/Model` with `Domain/Validator` and not with `Validation/Validator` as required.

11.5. Action Validation

The property validator is responsible for the property itself, the object validator takes care of the object and the action validator is triggered as soon as the action of a controller is executed.

Sometimes it is not necessary to do the same validation at every action. For example, when a new user is created you typically validate his/her email address, but if this address cannot be changed when you edit the user record later, you do not want or need to validate the address again.

In order to tackle this, an annotation can be set which specifies the validation:

```
/**
 * add action - adds a blog to the repository
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @validate $blog \Lobacher\Simpleblog\Validation\Validator\SpecialValidato
 */
 public function addAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog) {
 ...
```

As an exercise, the following example checks if the title entered matches an existing Facebook user. We will use a so-called service class to perform the check.

First, let’s create a new file `FacebookValidator.php`, stored in the directory `typo3conf/ext/simpleblog/Classes/Validation/Validator`:

```
<?php
namespace Lobacher\Simpleblog\Validation\Validator;

class FacebookValidator extends \TYPO3\CMS\Extbase\Validation\Validator\Abst

   /**
    * API Service
    *
    * @var \Lobacher\Simpleblog\Service\ExternalApiService
    * @inject
    */
   protected $apiService;

   /**
    * Object Manager
    *
    * @var \TYPO3\CMS\Extbase\Object\ObjectManagerInterface
    * @inject
    */
   protected $objectManager;
   /**
    * Validates the given value
    *
    * @param mixed $value
    * @return bool
    */
   protected function isValid($value) {
      $apiValidationResult = $this->apiService->validateData($value);
      $success = TRUE;
      if ($apiValidationResult['title']) {
         $error = $this->objectManager->get('TYPO3\\CMS\\Extbase\\Validation
            $apiValidationResult['title'], 1389545453);
         $this->result->forProperty('title')->addError($error);
         $success = FALSE;
      }
      return $success;
   }
}
?>
```

By using dependency injection we can fetch our service class (which we will create in a minute) and the so-called object manager. This allows us to load the error class in the method `isValid()`. Then we pass the object to the method `validateData()` of the API service class inside the method `isValid()` in order to check the validity. In theory other systems such as an Enterprise Resource Planning (ERP) system could be included at this point.

In the case of an error an error object is created and a `FALSE` response is returned. Let’s turn towards the service class, which we store as a new file `ExternalApiService.php` in the directory `typo3conf/ext/simpleblog/Classes/Service/`:

```
<?php
namespace Lobacher\Simpleblog\Service;

class ExternalApiService implements \TYPO3\CMS\Core\SingletonInterface {

    /**
     * @param \Lobacher\Simpleblog\Domain\Model\blog $blog
     * @return array
     */
    public function validateData(\Lobacher\Simpleblog\Domain\Model\blog $blo
        $errors = array();

        $fb = file_get_contents('http://graph.facebook.com/' . preg_replace(
            stream_context_create(
                array(
                    'http' => array(
                        'ignore_errors' => true
                    )
                )
            ));
        $fb = json_decode($fb, true);
        if (!$fb['error']) {
            $errors['title'] = 'Titel exists as an user at Facebook (ID = '.
        }
        return $errors;
    }

}
?>
```

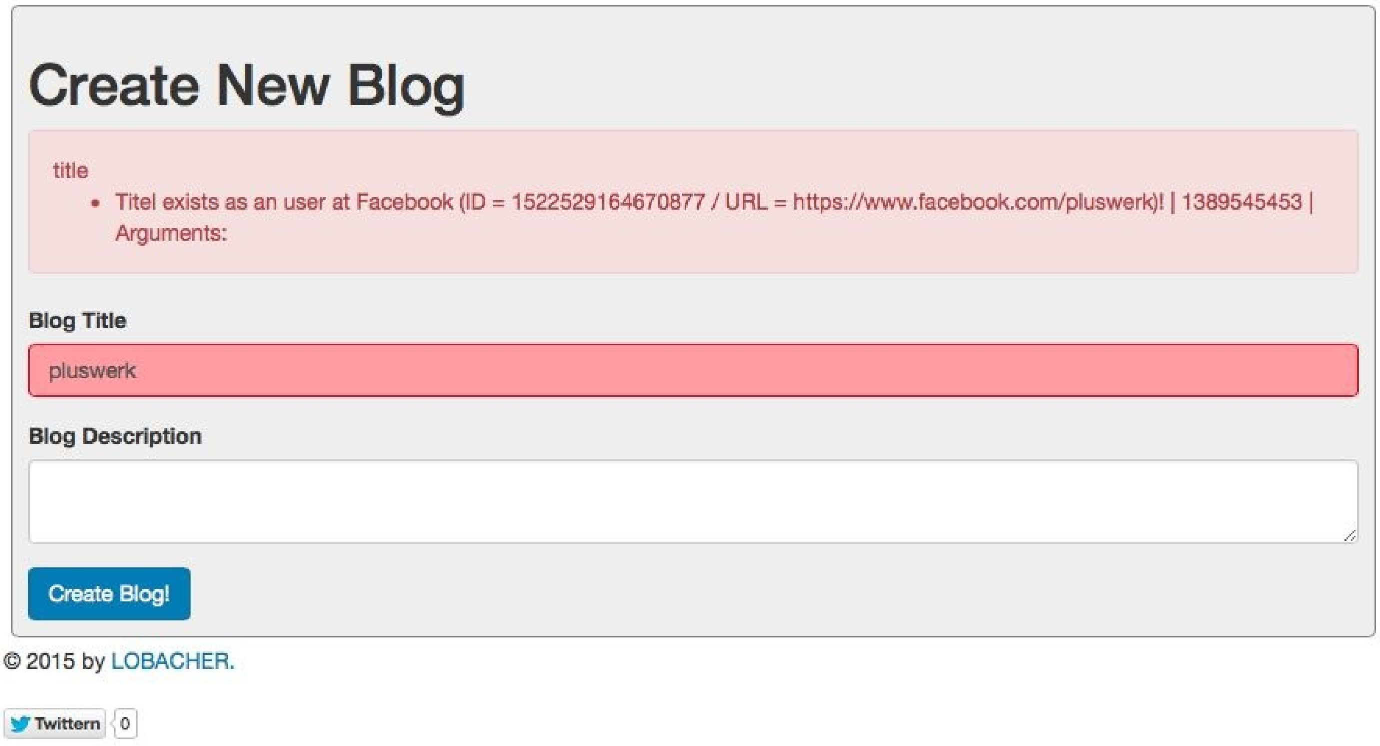
The method parameter specifies that we explicitly expect a Blog. Alternatively we could also build this in a more generic way and merely request an entity object.

The validation method executes an HTTP request from Facebook’s Graph API. If the API returns something we can assume that the name exists. If `NULL` is returned from the external API there is no Facebook username that matches the Blog title entered and our validation returns `TRUE`.

Finally, we have to attach the validation to an action in the controller `typo3conf/ext/simpleblog/Classes/Controller/BlogController.php`. The action `addAction()` seems to be a good choice because it ensures that the validation only occurs when a new object is created.

```
/**
 * add action - adds a blog to the repository
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @validate $blog Lobacher.Simpleblog:Facebook
 */
public function addAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog) {
...
```

Figure 11.6. Error message, if user exists at Facebook



Create New Blog

title

- Titel exists as an user at Facebook (ID = 1522529164670877 / URL = https://www.facebook.com/pluswerk)! | 1389545453 | Arguments:

Blog Title

pluswerk

Blog Description

11.6. Error Display in the Form Field

We have already seen how to show an error message at the top of the form. However for usability purposes, it might be much nicer to display the message near the invalid field.

11.6.1. Option 1: In-house Means

Generally speaking the error message can be positioned at the element and if you look closely you can see an array that can be addressed directly:

```
<f:form.validationResults for="blog">
   <f:if condition="{validationResults.flattenedErrors.title}">Error in Titl
</f:form.validationResults>
```

The if-condition checks if a key title exists in the array `flattenedErrors`, which in turn is part of `validationResults`.

Our aim is to highlight the appropriate label of the form in the colour red if an error has occurred:

```
<label class="text-danger">Blog Title</label>
```

The problem though, is that we cannot include the `<f:form.validationResults>` ViewHelper in the `<label>` tag as this would cause a syntax error. The solution is to use an alternative notation of the ViewHelper: the *inline syntax*.

Almost every ViewHelper can be used in inline syntax. Instead of angle brackets, curly brackets are used as the starting delimiter followed, as usual, by the ViewHelper name. Parentheses are used instead of spaces and contain all of the options (comma-separated) with a colon between the option name and value.

The values have the following meanings:

`''String''`
: Equates to a string.

`"String"`
: Equates to a string, too.

`Object`
: This is obviously an object.

`Object.Property`
: Addresses a property of an object.

`'This is an {object value}'`
: Outputs a string which contains a value of an object.

Let’s try to write the ViewHelper as inline syntax:

```
<f:if condition="{validationResults.flattenedErrors.title}"> class="text-dan

// Inline Syntax
{f:if(condition:validationResults.flattenedErrors.title,then:' class="text-d
```

Once the inline-ViewHelper has been added to the label, the label will contain the CSS class `text-danger` if an error occurs. It is important to wrap the `<label>` tag with

`<form.validationResults>` ViewHelper or the error will not be imported and therefore not available.

```
<f:form.validationResults for="blog">
   <label{f:if(condition:validationResults.flattenedErrors.title,then:' clas
</f:form.validationResults>
```

11.6.2. Option 2: ViewHelper

Another, more convenient option, to highlight the label is to write your own ViewHelper. We will cover this in more detail in Chapter 13 but the following provides a quick glance at what is possible.

Firstly, create a new folder `ViewHelpers` under `Classes` and store a new file `HasErrorViewHelper.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers;

class HasErrorViewHelper extends \TYPO3\CMS\Fluid\ViewHelpers\Form\AbstractF

   public function initializeArguments() {
      $this->registerArgument('property', 'string', 'Name of object property
   }

   /**
    * @param string $then
    * @param string $else
    * @return string
    */
   public function render($then = '', $else = '') {
      $originalRequestMappingResults = $this->controllerContext->getRequest(
      $formObjectName = $this->viewHelperVariableContainer->get( 'TYPO3\\CMS
      $errors = $originalRequestMappingResults->forProperty( $formObjectName
      if ($errors->hasErrors() == 1) {
         return $then;
      } else {
         return $else;
      }

   }
}
?>
```

Returning to the template `typo3conf/ext/simpleblog/Resources/Partials/Blog/Form.html`, we have to ensure we are able to use our own ViewHelpers by adding a name space declaration at the top of the file:

```
{namespace tv = Lobacher\Simpleblog\ViewHelpers}
<h1>{headline}</h1>
...
```

Now, we can use the ViewHelper directly at the label:

```
<label{tv:hasError(property:'title',then:' class="text-danger"')}>Blog Title
```

The benefits are obvious:

- no wrapper ViewHelper is required;
- no complicated array is required, but properties can still be accessed as options; and,
- the ViewHelper is significantly more compact.

Please note that the ViewHelper is addressed by using the name space declared above, which is `<tv:hasError>` and not `<f:hasError>`.

Chapter 12. Relations

In the previous chapters we focussed on Blogs, now we will explore Posts. Initially, we defined a `1:n` relation in the Extension Builder, which means, a Blog has no or an arbitrary number of Posts. This chapter will answer the question, how relations are built.

12.1. Relation in Domain Model

First of all, it is worth having a look at the domain model, e.g. file `typo3conf/ext/simpleblog/Classes/Domain/Model/Blog.php`:

```
class Blog extends \TYPO3\CMS\Extbase\DomainObject\AbstractEntity {
...
    /**
     * The posts of a Blog
     *
     * @var \TYPO3\CMS\Extbase\Persistence\ObjectStorage<\Lobacher\Simpleblo
     * @lazy
     */
    protected $posts;

    /**
     * __construct
     *
     * @return Blog
     */
    public function __construct() {
        //Do not remove the next line: It would break the functionality
        $this->initStorageObjects();
    }

    /**
     * Initializes all ObjectStorage properties.
     *
     * @return void
     */
    protected function initStorageObjects() {
        /**
         * Do not modify this method!
         * It will be rewritten on each save in the extension builder
         * You may modify the constructor of this class instead
         */
        $this->posts = new \TYPO3\CMS\Extbase\Persistence\ObjectStorage();
    }

    /**
     * Adds a Post
     *
     * @param \Lobacher\Simpleblog\Domain\Model\Post $post
     * @return void
     */
    public function addPost(\Lobacher\Simpleblog\Domain\Model\Post $post) {
        $this->posts->attach($post);
    }

    /**
```

```
     * Removes a Post
     *
     * @param \Lobacher\Simpleblog\Domain\Model\Post $postToRemove The Post
     * @return void
     */
    public function removePost(\Lobacher\Simpleblog\Domain\Model\Post $postT
        $this->posts->detach($postToRemove);
    }

    /**
     * Returns the posts
     *
     * @return \TYPO3\CMS\Extbase\Persistence\ObjectStorage<\Lobacher\Simple
     */
    public function getPosts() {
        return $this->posts;
    }

    /**
     * Sets the posts
     *
     * @param \TYPO3\CMS\Extbase\Persistence\ObjectStorage<\Lobacher\Simpleb
     * @return void
     */
    public function setPosts(\TYPO3\CMS\Extbase\Persistence\ObjectStorage $p
        $this->posts = $posts;
    }

    ...
}
```

Firstly, it’s obvious that the type of the relation (1:n) is not marked as such. This is contrary to TYPO3 Flow, which uses an annotation of type `ORM\OneToMany(mappedBy="blog")`. The type of relation is stored in the *Table Configuration Array (TCA)*, which we will analyse later.

However we can see immediately that every relation is marked as an “Object Storage” of `\TYPO3\CMS\Extbase\Persistence\ObjectStorage<FQCN>` by using an annotation. The object storage equates the “SplObjectStorage”[34] to a large extent and is used to store objects. The constructor initialises the object storage.

In order to add an object to the object storage or to remove one from it, commands `attach` and `detach` exist, which are encapsulated by `addPost` and `removePost`. The setter and getter work adequately.

12.2. The Table Configuration Array (TCA)

A complete explanation of the TCA could fill an entire book of itself so we will only cover the parts, which are relevant for Extbase. A comprehensive introduction of the TCA, which is worth a read, can be found in the TYPO3 Core Documentation.[35]

The Table Configuration Array (also known as `$TCA` or `TCA`) is a global array inside TYPO3, which extends the database table definition far in excess of the possibilities of SQL and also adds metadata. The most important aim of the TCA is the provision of table definitions, which can be edited in the backend of TYPO3.

Database tables without a record in TCA are invisible in a manner of speaking. The TCA definition is (among other things) responsible for the following areas:

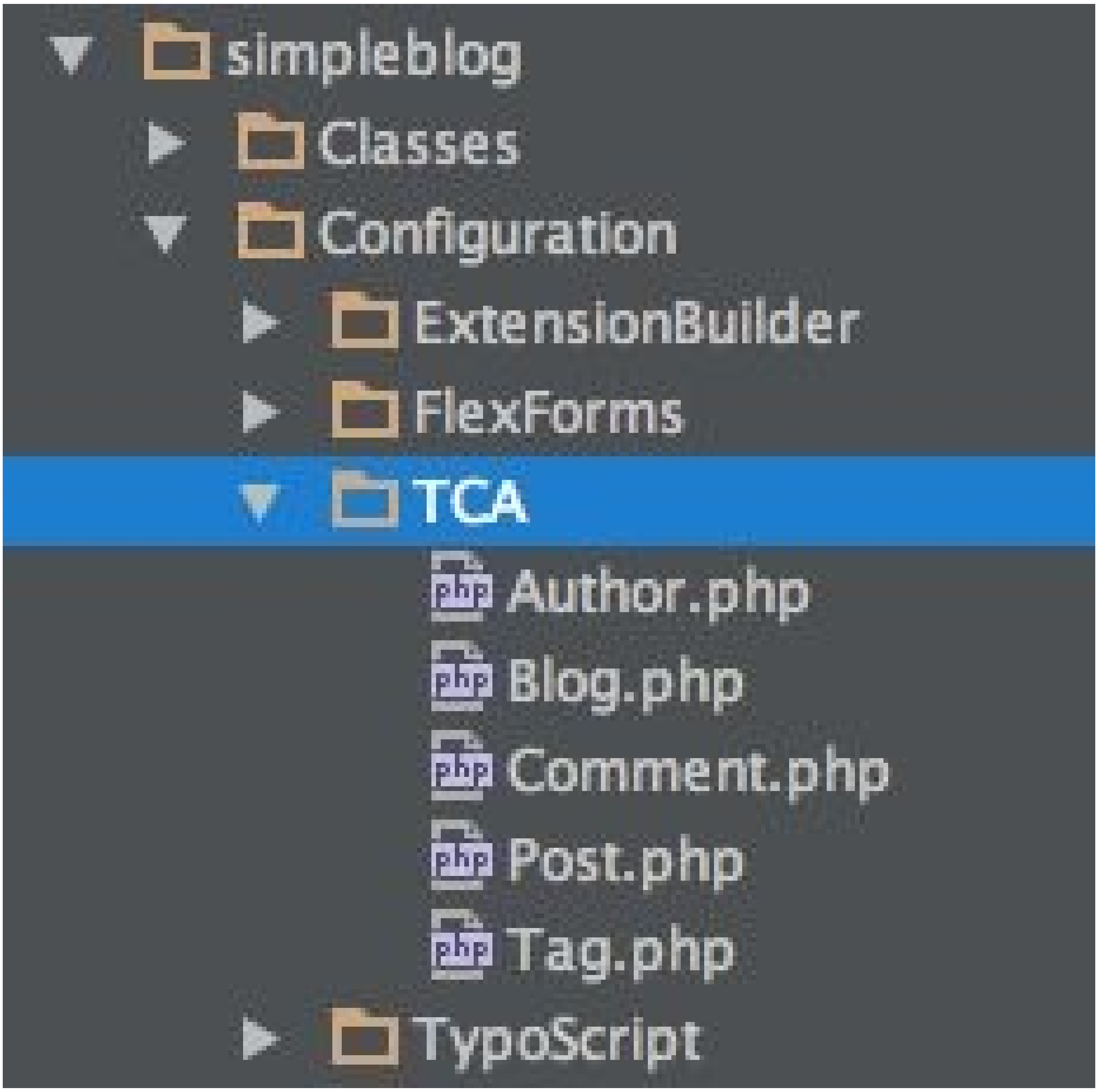
- relations between tables
- configuration, which fields should be displayed in the backend and their layout
- how the fields should be validated

The array can be extended and manipulated without any limitations by custom extensions. This means, extensions can add their own fields to existing tables, manipulate the configuration of existing fields or introduce new fields and tables.

Per convention, the TCA of extensions resides in directory `Configuration/TCA/` – for example `EXT:sys_note/Configuration/TCA/sys_note.php` or `typo3conf/ext/simpleblog/Configuration/TCA/Blog.php`.

Let’s have a look at the folder with the same name of our extension:

Figure 12.1. TCA directory structure



Location of TCA definitions

The official specification requires developers to store the TCA in a folder called `Configuration/TCA/`. A good example is the system extension `frontend`: all TCA files are named as the appropriate database table, e.g. `backend_layout.php` and they simply return the array of the TCA:

```
<?php
return array(
    'ctrl' => array(
        ...
    ),
    'interface' => array(
        ...
    ),
    'columns' => array(
        'title' => array(
            ...
        ),
        ...
    ),
    'types' => array(
        ...
    )
);
```

First of all, let’s open file `ext_tables.php` because the TCA will be included by a specific call in this file.

```
$TCA['tx_simpleblog_domain_model_blog'] = array(
    'ctrl' => array(
        'title' => 'LLL:EXT:simpleblog/Resources/Private/Language/locallang_
        'label' => 'title',
        'tstamp' => 'tstamp',
        'crdate' => 'crdate',
        'cruser_id' => 'cruser_id',
        'dividers2tabs' => TRUE,

        'versioningWS' => 2,
        'versioning_followPages' => TRUE,
        'origUid' => 't3_origuid',
        'languageField' => 'sys_language_uid',
        'transOrigPointerField' => 'l10n_parent',
        'transOrigDiffSourceField' => 'l10n_diffsource',
        'delete' => 'deleted',
        'enablecolumns' => array(
            'disabled' => 'hidden',
            'starttime' => 'starttime',
            'endtime' => 'endtime',
        ),
        'searchFields' => 'title,description,image,posts,',
        'dynamicConfigFile' => \TYPO3\CMS\Core\Utility\ExtensionManagementUt
        'iconfile' => \TYPO3\CMS\Core\Utility\ExtensionManagementUtility::ex
    ),
);
```

This code snippet (which exists for every domain object) creates the skeletal structure of the TCA – the “control section” (based on the key `ctrl`). The sub key `dynamicConfigFile` includes file `typo3conf/ext/simpleblog/Resources/Configuration/TCA/Blog.php` so please open this file too.

```
$TCA['tx_simpleblog_domain_model_blog'] = array(
    ...
    'columns' => array(
        ...
        'title' => array(
            'exclude' => 0,
                'label' => 'LLL:EXT:simpleblog/Resources/Private/Language/locall
                'config' => array(
                    'type' => 'input',
                    'size' => 30,
                    'eval' => 'trim,required'
                ),
            ),
            ...
            'posts' => array(
                'exclude' => 0,
                'label' => 'LLL:EXT:simpleblog/Resources/Private/Language/locall
                'config' => array(
                    'type' => 'inline',
                    'foreign_table' => 'tx_simpleblog_domain_model_post',
                    'foreign_field' => 'blog',
                    'maxitems'      => 9999,
                    'appearance' => array(
                        'collapseAll' => 0,
                        'levelLinksPosition' => 'top',
                        'showSynchronizationLink' => 1,
                        'showPossibleLocalizationRecords' => 1,
                        'showAllLocalizationLink' => 1
                    ),
                ),
            ),
        ),
);
```

The (shortened) code representation above shows that for every database table field a configuration exists – including internal fields, which are required by TYPO3.

At one point, field properties of the domain object follow, such as `title`. It is clearly visible that the field shows a configuration section labelled `config`. One of the elements in this section is the key `eval`, which forces a validation, if the field contains some data (`required`). This is the second validation in TYPO3, besides the `@validate NotEmpty` annotation, as mentioned earlier. It ensures that a value really exists.

Low-level database access

It is extremely important that both components – the validation annotation as well as the TCA – are in sync. If, for example, the `NotEmpty` annotation is removed but `required` in TCA remains, Extbase can store a domain object with an empty value but not the backend because the latter accesses the TCA.

The second half of the TCA shown above implements the relation. The type of the relation has been specified as `'type' => 'inline'`, which is a special type of relation. It assumes, that objects connected via this relation, are always “subordinated” to a parent object and that their existence would be pointless, without a parent object. The deletion of the parent object could also result in the deletion of all child objects. This is a valid assumption in our case because a Post without a Blog does not make any sense.

If you want to prevent this from happening, simply change the type to `select`. In this case, there will be a second object “connected” to the first one but without a parent/child relationship.

At the same time, type `inline` also makes sure, that child objects can be created inside of parent objects, without the need to reload the backend. This behaviour is called *Inline Relational Record Editing* (IRRE).

The two following configurations `foreign_table` and `foreign_field` configure the relation even further. The table, to which a relation exists, is stated in `foreign_table` and the field, which contains the identifier of the source table, is stated in `foreign_field`.

Technically speaking, with a `1:n` relation between Blog and Post, Extbase stores the UID of the Blog in field `blog` of table `tx_simpleblog_domain_model_post` and the number of Posts as `posts` in `table tx_simpleblog_domain_model_blog`.

12.3. The CRUD Process of Posts

This section explains how to implement the Posts by using a similar CRUD process as we did for the Blogs before. Let’s start with the single view of Post.

12.3.1. Preparation

In order to create Posts, we need some actions. Therefore, edit file `typo3conf/ext/simpleblog/ext_localconf.php`:

```
\TYPO3\CMS\Extbase\Utility\ExtensionUtility::configurePlugin(
    'Lobacher.' . $_EXTKEY,
    'Bloglisting',
    array(
        'Blog' => 'list,addForm,add,show,updateForm,update,deleteConfirm,del
        'Post' => 'addForm,add,show,updateForm,update,deleteConfirm,delete',
    ),
    // non-cacheable actions
    array(
        'Blog' => 'list,addForm,add,show,updateForm,update,deleteConfirm,del
        'Post' => 'addForm,add,show,updateForm,update,deleteConfirm,delete',
    )
);
```

As we can see, all actions have been added, which are required for the whole CRUD process. Only the `list` action of the Post controller has been left out, which would list the Posts in the single view of a Blog.

Open the template of the show action `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/Show.html`:

```
...
    <dd>{blog.description}</dd>
</dl>

<f:if condition="{blog.posts}">
    <ul class="list-group">
        <f:for each="{blog.posts}" as="post">
            <li class="list-group-item">{post.title}
                <f:link.action action="deleteConfirm" controller="Post" argu
                <f:link.action action="updateForm" controller="Post" argumen
                <f:link.action action="show" controller="Post" arguments="{b
            </li>
        </f:for>
    </ul>
</f:if>

<f:link.action action="addForm" controller="Post" arguments="{blog:blog}" cl

<br /><br />

<f:link.action action="list" class="btn btn-primary">Back to Blog List</f:li
...
```

If we use a `<f:if>` ViewHelper, we can check to see if the current Blog contains any Posts. The property `blog.posts` exists for exactly this reason. If there are posts then they get listed in a `<for each…>` loop. It is important that we know to which Blog a Post belongs, so the option `arguments` contains the current Post, which can be addressed by `post`. We also need to name a controller in the link to the Post because we are still in the Blog controller context.

The same applies to the link to create new Posts but the option `arguments` does not require the `post:post` value.

12.3.2. Create Posts

To create new Posts, we need an action `addFormAction`, which will render the form.

Therefore, we need the following code inside the file `typo3conf/ext/simpleblog/Classes/Controller/PostController.php`:

```
<?php
namespace Lobacher\Simpleblog\Controller;

class PostController extends \TYPO3\CMS\Extbase\Mvc\Controller\ActionControl

        /**
         * postRepository
         *
         * @var \Lobacher\Simpleblog\Domain\Repository\PostRepository
         * @inject
         */
        protected $postRepository = NULL;

    /**
     * Persistence Manager
     *
     * @var \TYPO3\CMS\Extbase\Persistence\Generic\PersistenceManager
     * @inject
     */
    protected $persistenceManager;

    /**
     * addForm action - displays a form for adding a post
     *
     * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
     * @param \Lobacher\Simpleblog\Domain\Model\Post $post
     */
    public function addFormAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post = NULL) {
        $this->view->assign('blog',$blog);
        $this->view->assign('post',$post);
    }

}
```

In comparison to the method in the Blog controller with the same name, one important thing has changed: we have to pass-through the Blog object. This is why it appears as an annotation, as a parameter in the method signature and in the assignment at the view.

It is important that optional parameters are defined after mandatory elements in the method signature so that `$blog` comes first and `$post` as the second parameter after that.

This method is ready to render the `AddForm.html` template of the Post object now. Create a new directory `Post` inside `typo3conf/ext/simpleblog/Resources/Private/Templates` and a new file `AddForm.html` with the following content:

```
<f:layout name="default" />

<f:section name="content">

<f:render partial="Post/Form" arguments="{headline:'Create new post',action:

</f:section>
```

As you can see, the code has been rewritten so that a Post can be created.

As we will re-use the layout, we can turn to the partial straight away. Create a new folder `typo3conf/ext/simpleblog/Resources/Private/Partials/Post` and a file `Form.html` inside, with the following content:

```
<h1>{headline}</h1>

<f:form action="{action}" object="{post}" name="post" arguments="{blog:blog}

    <div class="form-group">
        <label>Post Title</label>
        <f:form.textfield property="title" class="form-control" />
    </div>

    <div class="form-group">
        <label>Post Content</label>
        <f:form.textarea property="content" class="form-control" />
    </div>

    <f:form.submit value="{submitmessage}" class="btn btn-primary" />

</f:form>
```

This time, no major changes happened (compared to the Blog creation), except the fact that we pass through the Blog in the ViewHelper `form` by using the option `arguments`.

To catch the form data after a submit, we also need an `addAction` in the Post controller:

```
/**
 * add action - adds a post to the repository
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @param \Lobacher\Simpleblog\Domain\Model\Post $post
 */
public function addAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post) {
    $post->setPostdate(new \DateTime());

    //$this->postRepository->add($post);

    $blog->addPost($post);
    $this->objectManager->get( 'Lobacher\\Simpleblog\\Domain\\Repository\\Bl
    $this->redirect('show','Blog',NULL,array('blog'=>$blog));
}
```

First, we accept the Blog, which has been passed-through, as well as the new Post, in the method signature. We can set the current time stamp `postdate` at the beginning of the method or in the constructor of the domain object.

If could expect that everything can be done by writing `$this->postRepository->add($post)`. This way, the Post can be stored in the repository but does not have a connection to the current Blog. Therefore we commented out this line of code again.

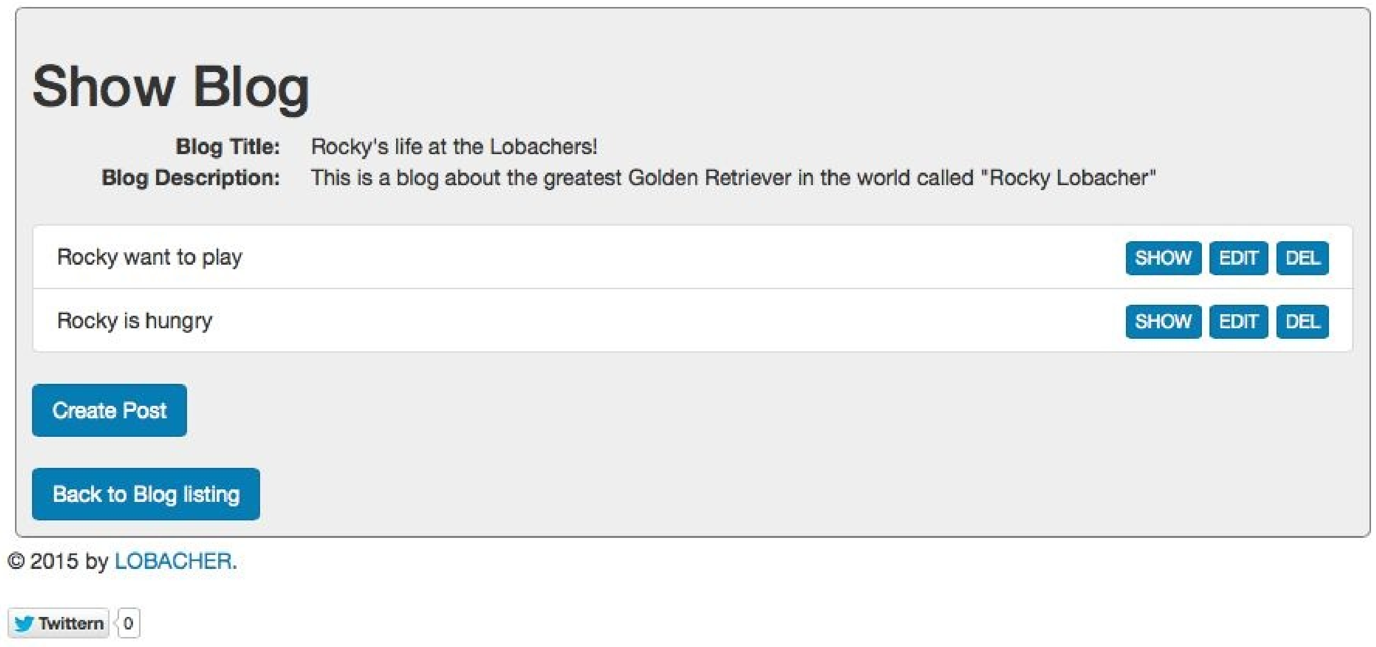
It is necessary to assign the Post to a Blog. This happens by adding `$blog->addPost($post);`. This was a requirement in versions of TYPO3 CMS prior to 6.1. However since then, you have to update the object explicitly, with received child objects (or remove child objects).

In theory, we can fetch the Blog repository by using Dependency Injection but this would do this for all actions. Instead, we should get the repository only at this single point, which can be achieved with the “Object Manager” (it is loaded in the controller by default). The main purpose of the Object Manager is to load or create an object.

Method `get()` loads the Blog repository and the update is done by executing the `update()` method.

Finally, we need to redirect the user back to the action `show` of the Blog controller and pass the appropriate Blog as a parameter (this is the fourth parameter). The third parameter names the extension but due to the fact that we do not leave it, we set this to `NULL`.

Figure 12.2. List of Posts of a Blog



Show Blog

Blog Title:	Rocky's life at the Lobachers!
Blog Description:	This is a blog about the greatest Golden Retriever in the world called "Rocky Lobacher"

Rocky want to play

Rocky is hungry

© 2015 by LOBACHER.

Exercise

As an exercise, try to implement the remaining steps of the CRUD process yourself and compare the results at the end (or after each step) with the following sections. Based on the knowledge you gained so far, this should be straightforward.

12.3.3. Read Posts

The implementation of the single view of a post is very similar to the Blog. The appropriate action in file `ext_localconf.php` already exists so we do not need to worry about this.

Let’s look at the Post controller `typo3conf/ext/simpleblog/Classes/Controller/PostController.php` and add the following action:

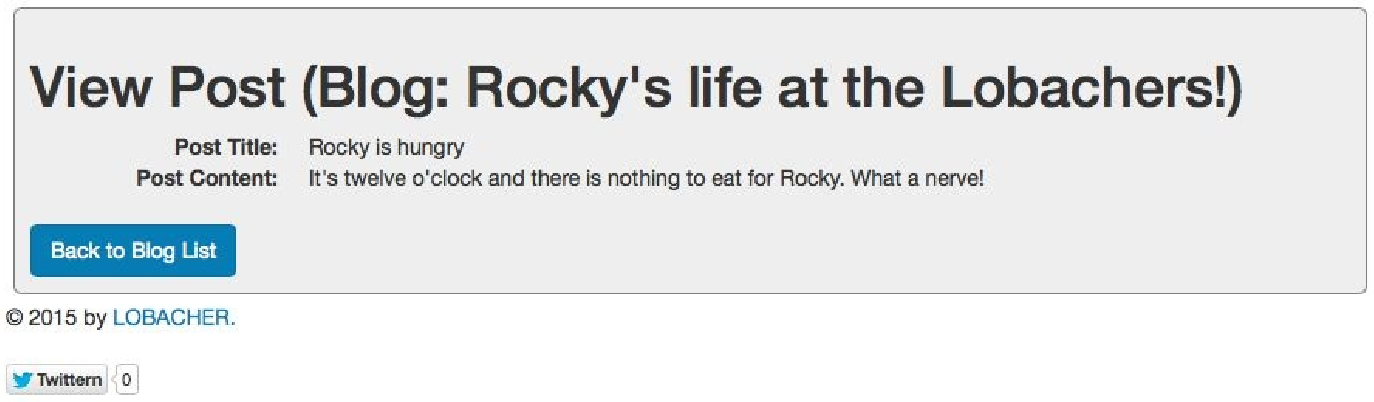
```
/**
 * show action - displays a single post
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @param \Lobacher\Simpleblog\Domain\Model\Post $post
 */
public function showAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post) {
    $this->view->assign('blog',$blog);
    $this->view->assign('post',$post);
}
```

As before, we have to pass through the Blog because we would like to redirect to the action `show` of the `Blog` controller later.

Create file `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/Show.html` with the following content:

```
<f:layout name="default" />

<f:section name="content">

<h1>View Post (Blog: {blog.title})</h1>

<dl class="dl-horizontal">
    <dt>Post Title:</dt>
    <dd>{post.title}</dd>
    <dt>Post Content:</dt>
    <dd>{post.content}</dd>
</dl>

<f:link.action action="show" controller="Blog" arguments="{blog:blog}" class

</f:section>
```

Figure 12.3. Single view of a Post



View Post (Blog: Rocky's life at the Lobachers!)

Post Title:	Rocky is hungry
Post Content:	It's twelve o'clock and there is nothing to eat for Rocky. What a nerve!

© 2015 by LOBACHER.

12.3.4. Update Posts

Editing (updating) a post works the same way as the Blog. Open the Post controller:

```
/**
 * updateForm action - displays a form for editing a post
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @param \Lobacher\Simpleblog\Domain\Model\Post $post
 */
public function updateFormAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post) {
    $this->view->assign('blog',$blog);
    $this->view->assign('post',$post);
}

/**
 * update action - updates a post in the repository
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @param \Lobacher\Simpleblog\Domain\Model\Post $post
 */
public function updateAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post) {
    $this->postRepository->update($post);
    $this->redirect('show','Blog',NULL,array('blog'=>$blog));
}
```

We added both actions required at the same time. An interesting element is the line `$this->postRepository->update($post);`, which finds the record in the database automatically and updates it.

We also need a template `UpdateForm.html` in directory `typo3conf/ext/simpleblog/Resources/Private/Templates/Post`:

```
<f:layout name="default" />

<f:section name="content">

<f:render partial="Post/Form" arguments="{headline:'Update Post',action:'upd

</f:section>
```

12.3.5. Delete Posts

We can delete a post by using a two-step-process, including a confirmation page. Let's start with the Post controller again:

```
/**
 * deleteConfirm action - displays a form for confirming the deletion of a p
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @param \Lobacher\Simpleblog\Domain\Model\Post $post
 */
public function deleteConfirmAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post) {
    $this->view->assign('blog',$blog);
    $this->view->assign('post',$post);
}

/**
 * delete action - deletes a post in the repository
 *
 * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
 * @param \Lobacher\Simpleblog\Domain\Model\Post $post
 */
public function deleteAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post) {
    $blog->removePost($post);
    $this->objectManager->get( 'Lobacher\\Simpleblog\\Domain\\Repository\\Bl
    $this->postRepository->remove($post);
    $this->redirect('show','Blog',NULL,array('blog'=>$blog));
}
```

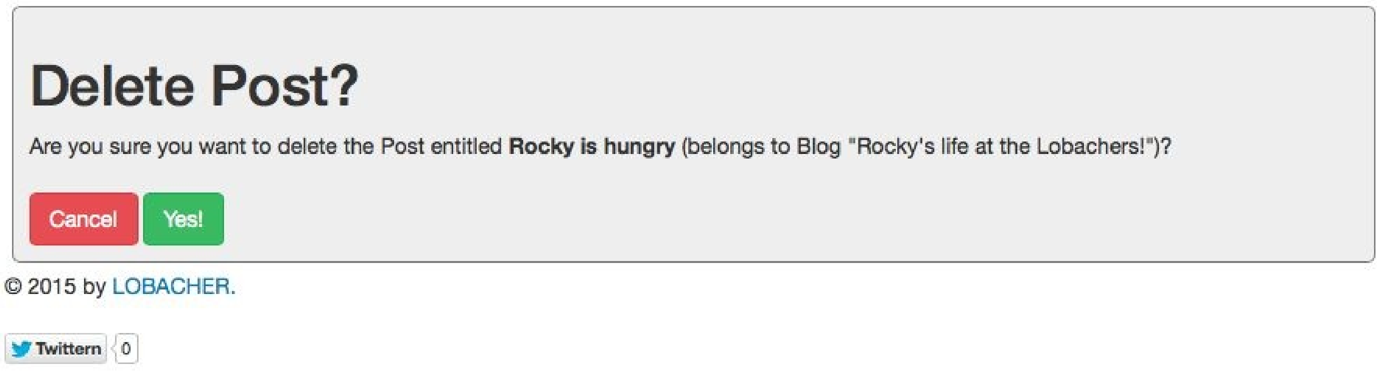
The `deleteAction` not only deletes the Post from the appropriate repository but also removes the relationship between Blog and Post. This is not required necessarily (Extbase takes care of this automatically) but proves a clean and professional style of software development.

Finally, we need a template for the confirmation page, which we can create under the name `DeleteConfirm.html` in directory `typo3conf/ext/simpleblog/Resources/Private/Templates/Post` with the following content:

```
<f:layout name="default" />

<f:section name="content">

<h1>Delete Post?</h1>

Are you sure you want to delete the Post entitled <strong>{post.title}</stro

<br /></br >
```

```
<f:link.action action="show" controller="Blog" arguments="{blog:blog}" class
<f:link.action action="delete" class="btn btn-success" arguments="{blog:blog

</f:section>
```

Figure 12.4. Deletion of a Post



Delete Post?

Are you sure you want to delete the Post entitled **Rocky is hungry** (belongs to Blog "Rocky's life at the Lobachers!")?

12.4. m:n Relations Using the Example of Tags

Relations of type `m:n` are implemented by using an intermediate table (in our case: `tx_simpleblog_post_tag_mm`) and managed automatically. Strictly speaking, the programming is nothing new, compared with the programming of the Posts by applying the CRUD rules. Extbase takes care of everything required to resolve these types of relationships.

However we will build a special feature into the code by abandoning the option to store the data into the repository and set the property of the relation directly.

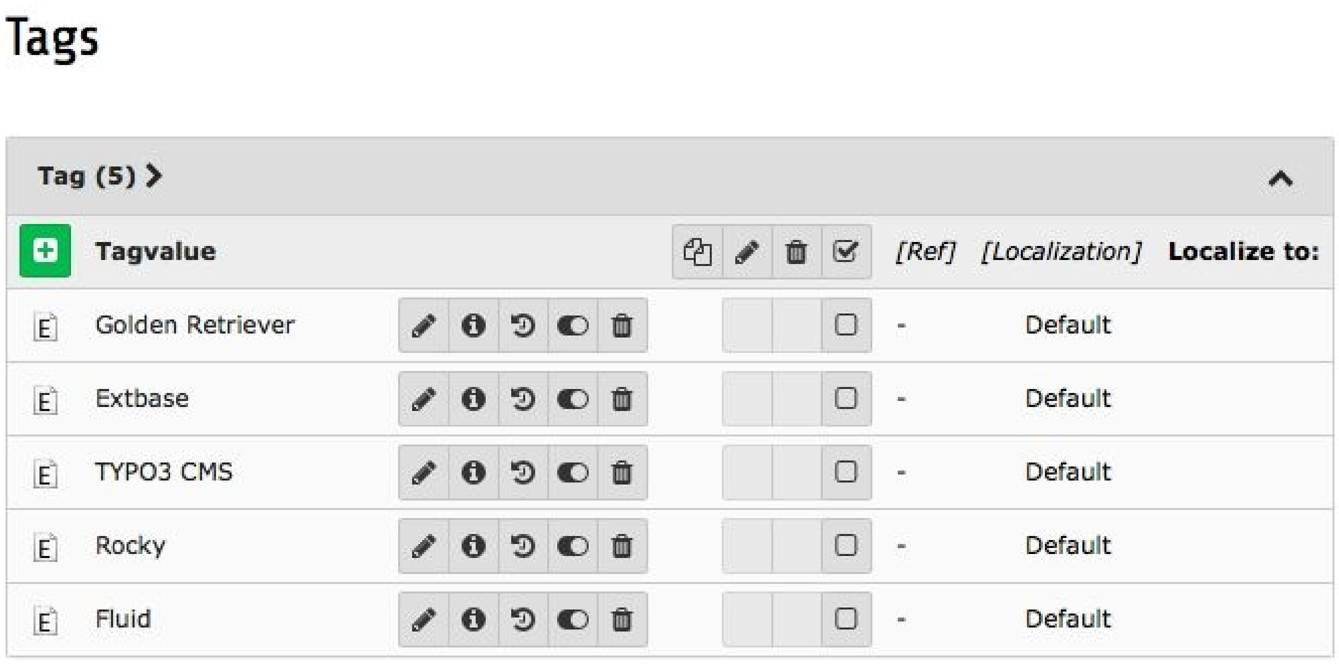
12.4.1. Creation of Tags in the Backend

As the first step, in the backend, we create the following five tags in the folder `Tags`, which we have created before (under `BlogData`):

- `Golden Retriever`
- `Extbase`
- `TYPO3 CMS`
- `Rocky`
- `Fluid`

The most suitable way to do this, is by using the module *List* and then by clicking the green plus symbol at the top of the page. Choose *Tag* in section *Simple Blog Extension*.

Figure 12.5. Creation of the five Tags



Tags

Tag (5)

Tagvalue		[Ref]	[Localization]	Localize to:
Golden Retriever		-		Default
Extbase		-		Default
TYPO3 CMS		-		Default
Rocky		-		Default
Fluid		-		Default

12.4.2. Repository for Tags

Tags should be listed when creating a new Post so that the user can select them straight away. First we have to retrieve them, which means that we should use a repository for this. Unfortunately, this repository does not exist yet so let's take care of that by creating a new file `typo3conf/ext/simpleblog/Classes/Domain/Repository/TagRepository.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\Domain\Repository;

class TagRepository extends \TYPO3\CMS\Extbase\Persistence\Repository {

}
?>
```

As you can see, the class is empty. This is not a problem because all functions required exist in the abstract class and secondly, we do not need any further functions at this point in time.

12.4.3. Post-Controller Adjustments

In order to show the Tags in the form where users can create new Posts, we have to retrieve them in the action method of the Post-Controller. Extend file `typo3conf/ext/simpleblog/Classes/Domain/Controller/PostController.php` as follows:

```
...
    public function addFormAction(
            \Lobacher\Simpleblog\Domain\Model\Blog $blog,
            \Lobacher\Simpleblog\Domain\Model\Post $post = NULL) {
        $this->view->assign('blog',$blog);
        $this->view->assign('post',$post);
        $this->view->assign('tags', $this->objectManager->get( 'Lobacher\\Si
    }
...
```

By using the Object-Manager, we can access the Tag repository and fetch all Tags with the `findAll` call. After that, we assign the Tags to the view by using the `assign` function.

We have decided to use this approach (rather than the `@inject` annotation in the class header) because we only need to access the Tag repository at this specific point in our extension (and later for a second time). If we had to access the repository in nearly all actions, the annotation approach would be the preferred option.

In order to allow users to edit Tags later on, this change must be implemented in method `updateFormAction()` of the Post controller as well. So, we have to extend file `typo3conf/ext/simpleblog/Classes/Domain/Controller/PostController.php`, too:

```
...
    public function updateFormAction(
            \Lobacher\Simpleblog\Domain\Model\Blog $blog,
            \Lobacher\Simpleblog\Domain\Model\Post $post = NULL) {
        $this->view->assign('blog',$blog);
        $this->view->assign('post',$post);
        $this->view->assign('tags', $this->objectManager->get( 'Lobacher\\Si
    }
...
```

12.4.4. Templates and Partials Adjustments

Let’s continue with the appropriate template `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/AddForm.html`, where we pass the Tags to the partial:

```
...
<f:render partial="Post/Form" arguments="{headline:'Create new Post',action:
...
```

As well as in file `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/UpdateForm.html`, where we pass the Tags to the partial, too:

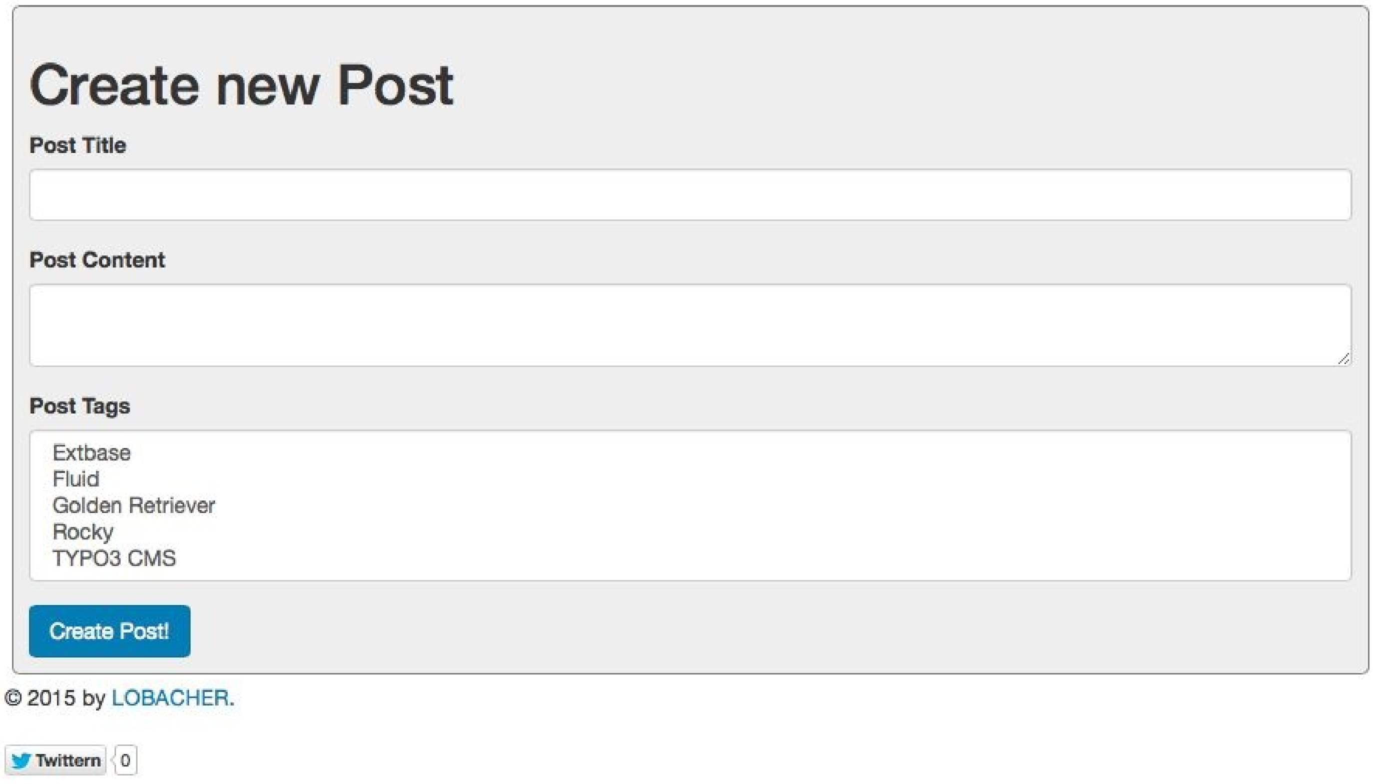
```
...
<f:render partial="Post/Form" arguments="{headline:'Create new Post',action:
...
```

Adding `tags:tags` here is sufficient to access the Tags in the partial `typo3conf/ext/simpleblog/Resources/Private/Partials/Post/Form.html`:

```
...
    <div class="form-group">
        <label>Post Tags</label>
        <f:form.select options="{tags}" optionLabelField="tagvalue" property
    </div>

    <f:form.submit value="{submitmessage}" class="btn btn-primary" />
...
```

Finally, the list of tags is shown.

Figure 12.6. List Tags in Post form


Create new Post

Post Title

Post Content

Post Tags

Extbase
Fluid
Golden Retriever
Rocky
TYPO3 CMS

The only minor issue is that the list is not sorted. We can easily correct this by adding the following line to the Tag repository:

```
...
class TagRepository extends \TYPO3\CMS\Extbase\Persistence\Repository {

    protected $defaultOrderings = array('tagvalue' => \TYPO3\CMS\Extbase\Per

}
...
```

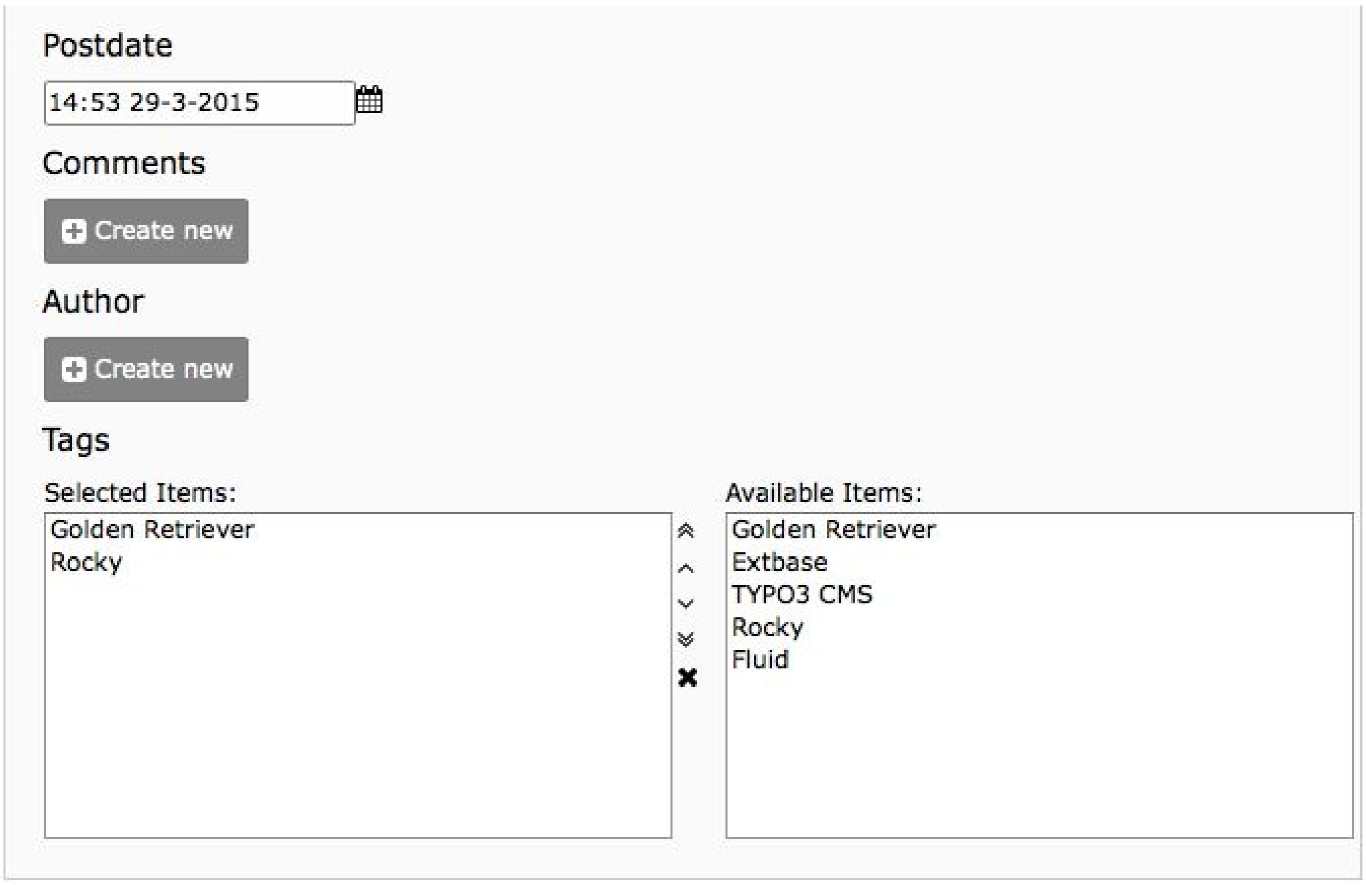
Property `defaultOrderings` is a shortcut to avoid writing your own repository method, which would only sort the data. Additionally, the property is valid for all query methods, until it is explicitly overwritten by `$query->setOrderings()`.

Finally, we have to adjust method `addAction` in Post controller in order to store the tags.

But is this really required? If we look at the partial more closely, we can see that the Tags are merely properties of the objects – the same as `title` or `content`. We do not need to make them persistent because this happens with the object automatically. This is exactly what Extbase does with a property, which contains a relation.

This means, we are already finished. You can verify this by accessing the data set in the backend: all selected Tags should be visible.

Figure 12.7. Backend view of the Post data set



Postdate

14:53 29-3-2015

Comments

Create new

Author

Create new

Tags

Selected Items:

Golden Retriever
Rocky

Available Items:

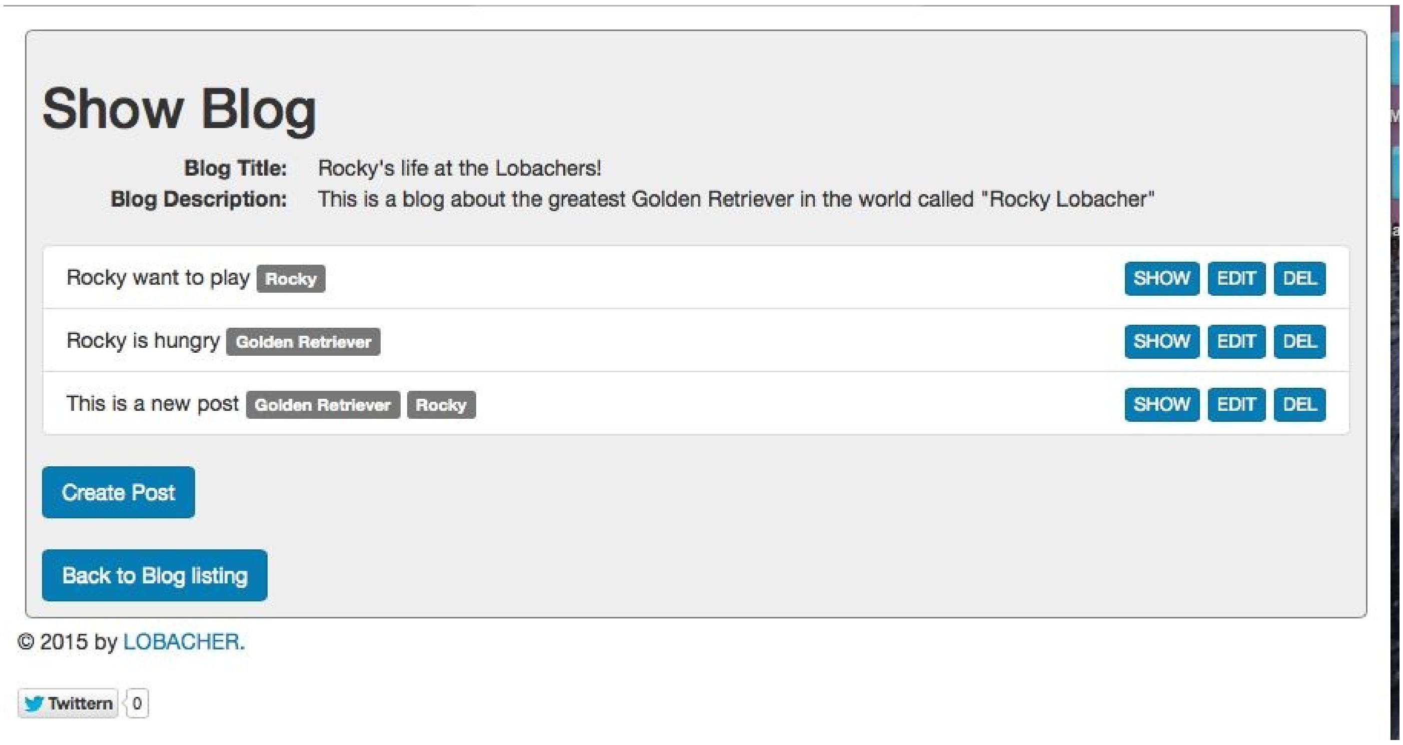
Golden Retriever
Extbase
TYPO3 CMS
Rocky
Fluid

12.4.5. Show Tags in List

In order to show the tags, we only need to read the property `post.tags`. This happens in file `typo3conf/ext/simpleblog/Resource/Private/Templates/Blog/Show.html`.

```
...
            <li class="list-group-item">{post.title}
                <f:for each="{post.tags}" as="tag">
                    <span class="label label-default">{tag.tagvalue}</span>
                </f:for>
                <f:link.action action="deleteConfirm" controller="Post" argu
...
```

Figure 12.8. Tags are shown at the Post List



Show Blog

Blog Title: Rocky's life at the Lobachers!
Blog Description: This is a blog about the greatest Golden Retriever in the world called "Rocky Lobacher"

Rocky want to play Rocky SHOW EDIT DEL
Rocky is hungry Golden Retriever SHOW EDIT DEL
This is a new post Golden Retriever Rocky SHOW EDIT DEL

Create Post

Back to Blog listing

© 2015 by LOBACHER.

Twittern 0

Attribute `iteration` in for-ViewHelper

If we would output each Tag as text and all Tags separated by a delimiter symbol (e.g. comma or slash), this would result in a trailing symbol at the end of the list. The attribute `iteration` of the `<f:for>` ViewHelper addresses this problem. This allows us to check which position of the iteration we are currently in. The following sub-properties exist: `isFirst` (the first element of the iteration), `isLast` (the last element of the iteration), `isEven` (number of current cycle is even), `isOdd` (number of current cycle is odd), `cycle` (current number of the cycle) and `index` (current index of the cycle, which is cycle minus 1).

```
<f:for each="{post.tags}" as="tag" iteration="count">
    {tag.tagvalue}{f:if(condition:count.isLast,then:'',else:',')}
</f:for>
```

We also have to add the list of tags to the template `Show.html` of course. Now we can edit the file `typo3conf/ext/simpleblog/Resource/Private/Templates/Post/Show.html`:

```
...
    <dd>{post.content}</dd>
    <dt>Post-Tags</dt>
    <dd><f:for each="{post.tags}" as="tag">
        <span class="label label-default">{tag.tagvalue}</span>
    </f:for></dd>
</dl>
...
```

12.5. The 1:1 Relation Using the Example of Authors

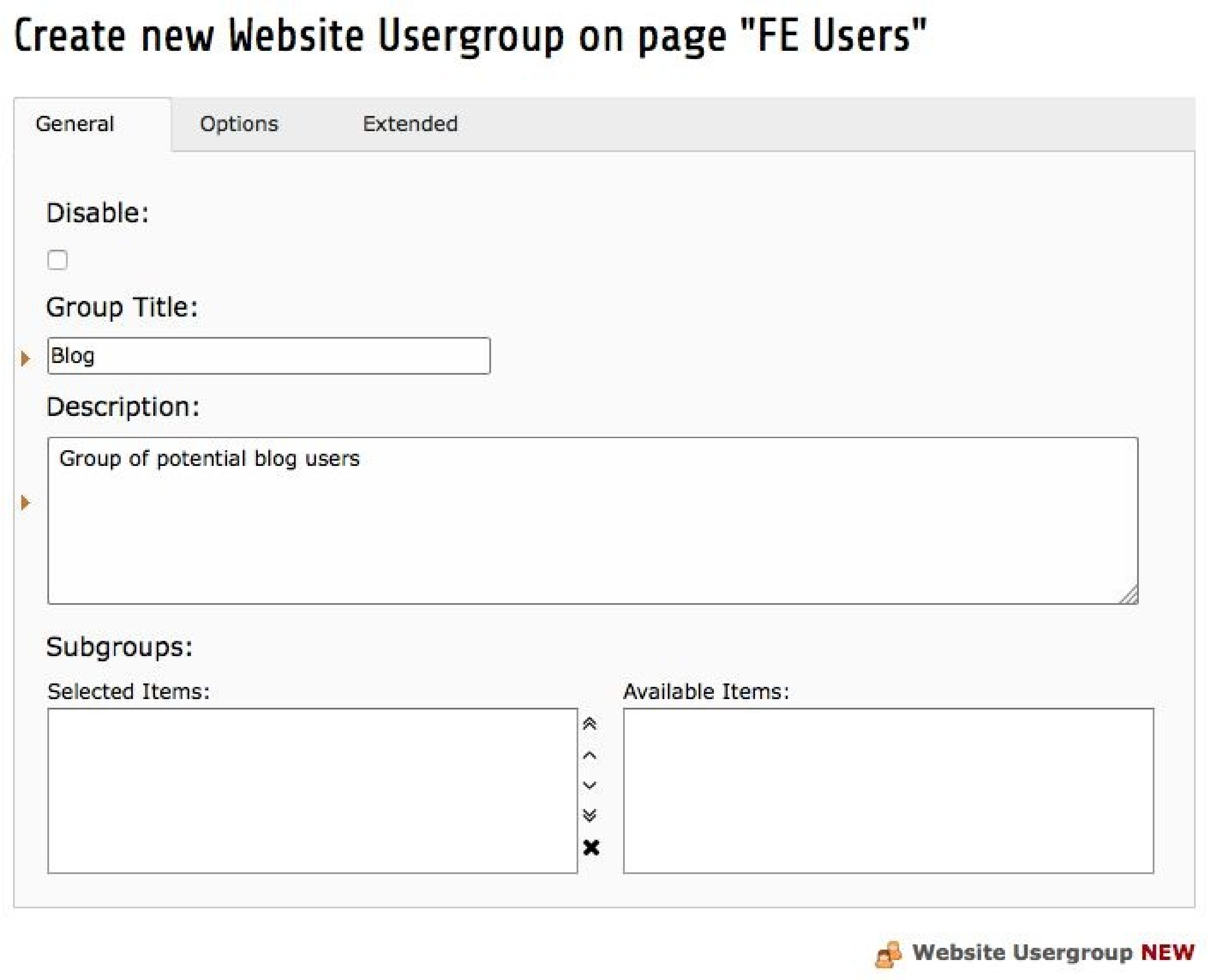
The next task will be to define an author of our Posts. This requires us to allow the selection of an another user? as the first step and to use the currently logged-in frontend user as the second step.

12.5.1. Creation of Frontend Users and Groups

The preliminary work required is to configure the system so that different? user accounts exist. Due to the fact that a frontend user must be assigned to at least a frontend usergroup, let's start with the group.

Create a new folder in the page tree where users and groups will be stored and name it `FE Users`. In this folder, create a new usergroup `Blog`.

Figure 12.9. Create new frontend usergroup



Create new Website Usergroup on page "FE Users"

Disable: ☐

Group Title: Blog

Description: Group of potential blog users

Subgroups:

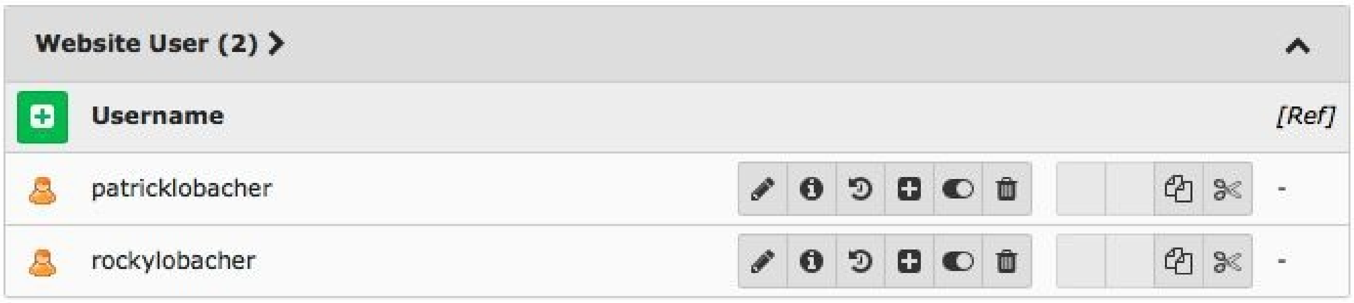
Selected Items:

Available Items:

After that, two frontend users are to be added. Use the following data:

- username: `rockylobacher`, password: `secretpassword`, usergroup: `Blog`, Name: `Rocky Lobacher`, email: `rocky@lobacher.de`
- username: `patricklobacher`, password: `secretpassword`, usergroup: `Blog`, Name: `Patrick Lobacher`, email: `patrick@lobacher.de`

Figure 12.10. Create frontend users


Website User (2)

Username	[Ref]
patricklobacher	-
rockylobacher	-

12.5.2. Link Domain Object author to fe_users Table

In principle we would like to have one of the frontend users as the author of a Post. This means, we have to link the domain model `Author` to the database table `fe_users`.

In order to do this, we have to adjust the TypoScript setup `typo3conf/ext/simpleblog/Configuration/TypoScript/Setup.txt`:

```
plugin.tx_simpleblog {
    ...
    persistence {
        storagePid = 7,8,6
        ...
        classes {
            ...
            Lobacher\Simpleblog\Domain\Model\Author {
                mapping {
                    tableName = fe_users
                    columns {
                        name.mapOnProperty = fullname
                    }
                }
            }
        }
    }
}
...
}
```

This adds the folder of the FE users (this is UID 6 in the case of the author of this book) as an additional value to the `storagePid`. In addition, a mapping has been introduced because the table can not be `tx_simpleblog_domain_model_author` but the default table `fe_users`. Finally, column `name` of table `fe_users` is mapped to the property `fullname` of the domain object.

Next, the TCA of the Post has to be updated. Edit the file `typo3conf/ext/simpleblog/Configuration/TCA/Post.php`:

```
        ...
        'author' => array(
            'exclude' => 0,
            'label' => 'LLL:EXT:simpleblog/Resources/Private/Language/locall
            'config' => array(
                'type' => 'select',
                'foreign_table' => 'fe_users',
                'minitems' => 0,
                'maxitems' => 1,
            ),
        ),
        ...
```

We change the type from `inline` (this allows us to create IRRE data sets) to `select` (an author is not a child-object of a Post) and enter the name of the table we would like to retrieve data from under `foreign_table`.

If we go to the backend and edit a Post, we can already see that there is a select box with frontend users. Unfortunately with usernames rather than full names so we have to update the TCA of the `fe_users` table, too. Add the following line to file `typo3conf/ext/simpleblog/ext_tables.php`:

```
$TCA['fe_users']['ctrl']['label'] = 'name';
```

12.5.3. Defining the Author When Creating and Editing a Post

We proceed in an analogical sense to the Tag in the previous section. First, we need an author repository, which we create as `typo3conf/ext/simpleblog/Classes/Domain/Repository/AuthorRepository.php` and with the following content:

```
<?php
namespace Lobacher\Simpleblog\Domain\Repository;

class AuthorRepository extends \TYPO3\CMS\Extbase\Persistence\Repository {

    protected $defaultOrderings = array('fullname' => \TYPO3\CMS\Extbase\Per

}
?>
```

Then, the Post controller `typo3conf/ext/simpleblog/Classes/Domain/Controller/PostController.php`:

```
    public function addFormAction(
            \Lobacher\Simpleblog\Domain\Model\Blog $blog,
            \Lobacher\Simpleblog\Domain\Model\Post $post = NULL) {
        ...
        $this->view->assign('tags', $this->objectManager->get( 'Lobacher\\Si
        $this->view->assign('authors', $this->objectManager->get( 'Lobacher\
    }

    public function updateFormAction(
        \Lobacher\Simpleblog\Domain\Model\Blog $blog,
        \Lobacher\Simpleblog\Domain\Model\Post $post) {
        ...
        $this->view->assign('tags', $this->objectManager->get( 'Lobacher\\Si
        $this->view->assign('authors', $this->objectManager->get( 'Lobacher\
}
...
```

Similar to the process before, we simply access the author repository and pass all objects to the view.

Now, template file `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/AddForm.html`:

```
...
<f:render partial="Post/Form" arguments="{headline:'Create new Post',action:
...
```

And template file `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/UpdateForm.html`:

```
...
<f:render partial="Post/Form" arguments="{headline:'Update Post',action:'upd
...
```

We only added `authors:authors` here.

Then, partial `typo3conf/ext/simpleblog/Resources/Private/Partials/Post/Form.html` should be slightly adjusted:

```
    ...
    <div class="form-group">
        <label>Post-Author</label>
         <f:form.select options="{authors}" optionLabelField="fullname" prep
    </div>
    ...
```

And finally the name of the author in template `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/Show.html`:

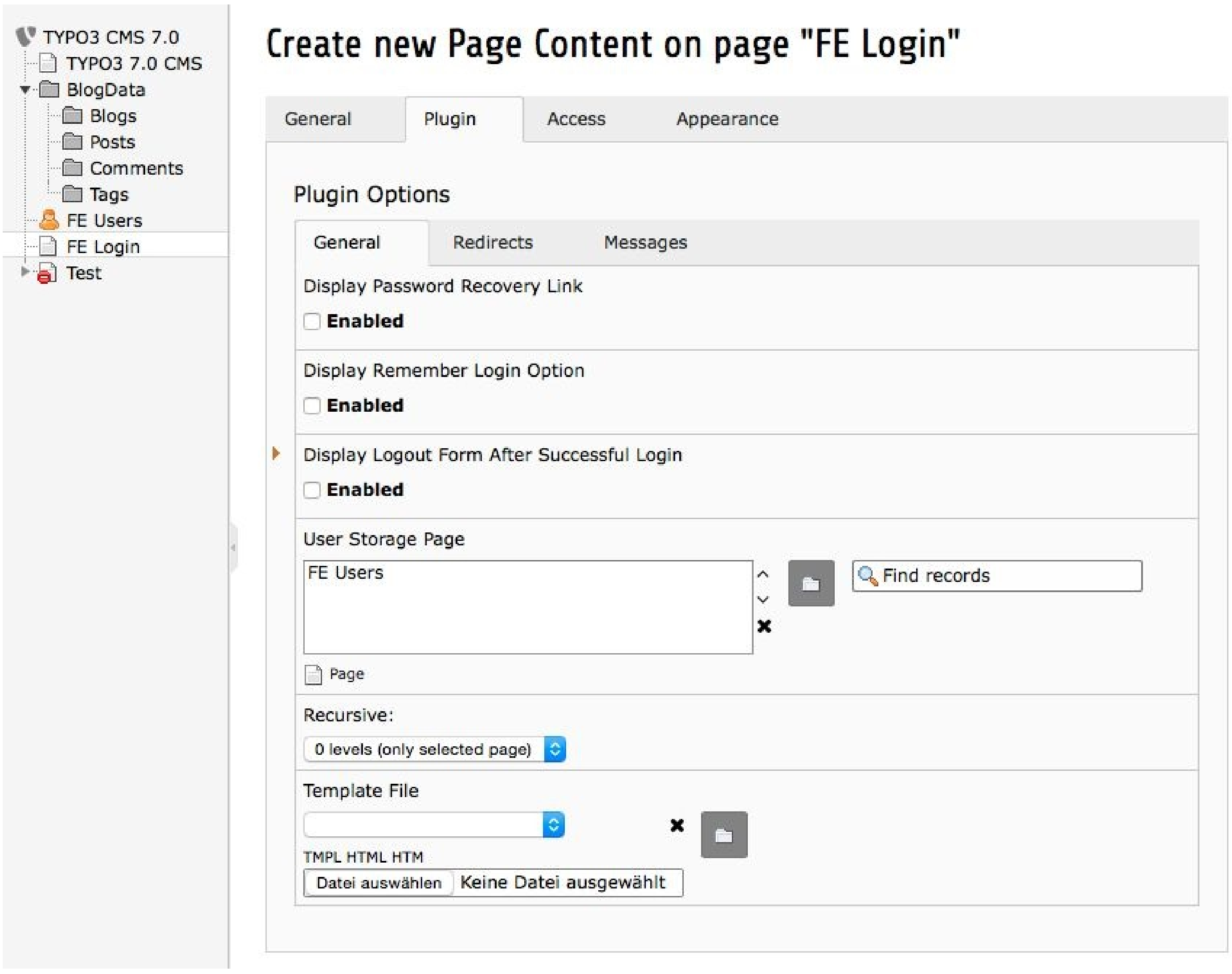
```
...
    <dd>{post.content}</dd>
    <dt>Post Author:</dt>
    <dd>{post.author.fullname} (Email: {post.author.email})</dd>
</dl>
...
```

12.5.4. Logged-in User as the Author

Instead of selecting the author manually, it is better to use the currently logged-in frontend user automatically. This requires a login form.

Create a new page in the backend and name it `FE Login` (in our example, the page UID is 12).

Figure 12.11. Configuration of the login form 1



Create new Page Content on page "FE Login"

Display Password Recovery Link
☐ Enabled

Display Remember Login Option
☐ Enabled

Display Logout Form After Successful Login
☐ Enabled

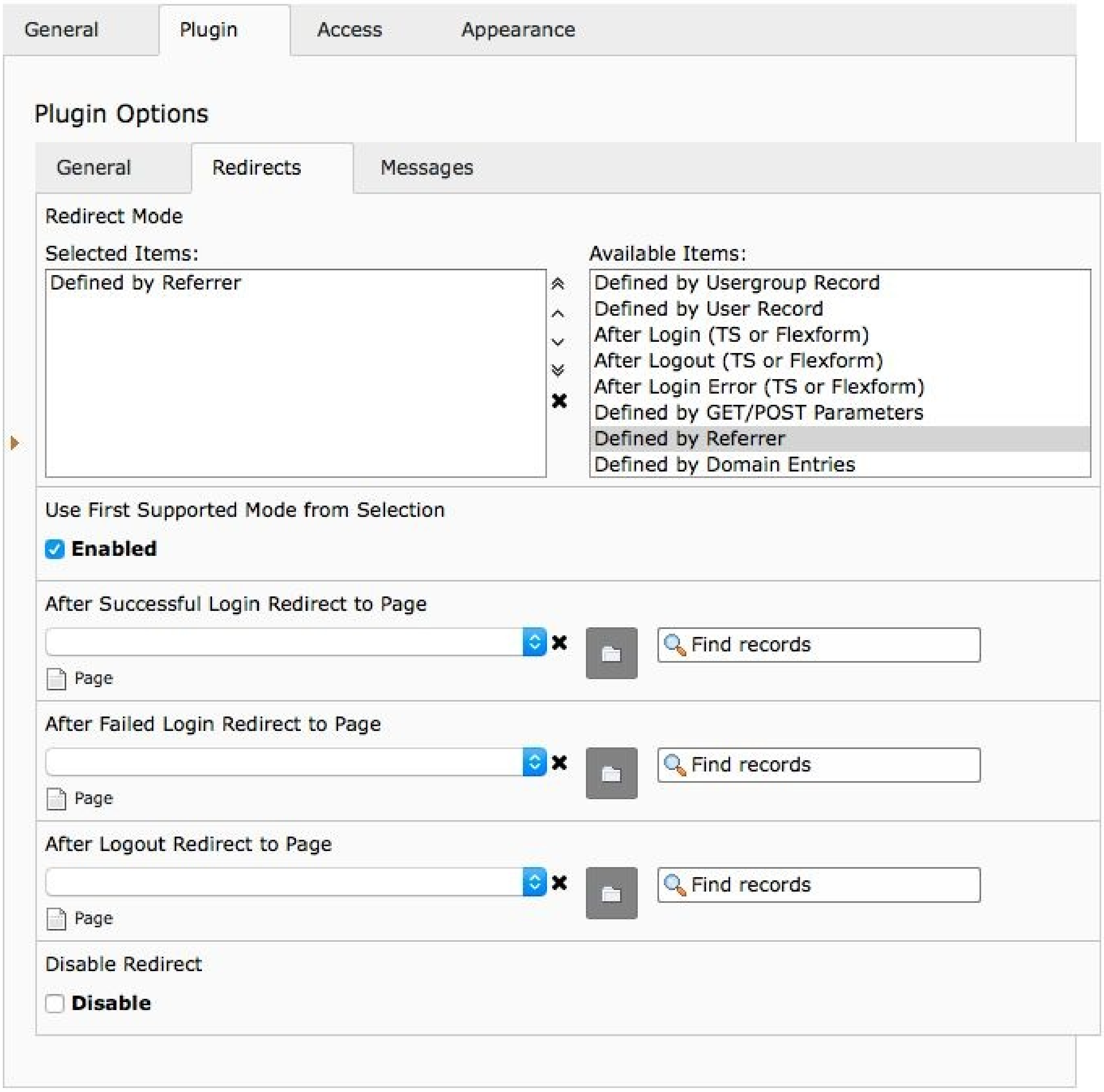
User Storage Page
FE Users

Recursive:
0 levels (only selected page)

Template File
TMPL HTML HTM
Datei auswählen Keine Datei ausgewählt

Then, add a new content element to this page and choose *Login Form* (tab *Form elements*). You can find some options under tab *Plugin*. One of them is the box *User Storage Page*, where we select the folder with the user accounts (in our example UID 6), which we created before. After that, we change to tab *Redirects* and choose *Defined by Referrer* under *Selected Items*.

Figure 12.12. Configuration of the login form 2


General | Plugin | Access | Appearance

Plugin Options

General | Redirects | Messages

Redirect Mode

Selected Items:
Defined by Referrer

Available Items:
Defined by Usergroup Record
Defined by User Record
After Login (TS or Flexform)
After Logout (TS or Flexform)
After Login Error (TS or Flexform)
Defined by GET/POST Parameters
Defined by Referrer
Defined by Domain Entries

Use First Supported Mode from Selection
☑ Enabled

After Successful Login Redirect to Page
Page

After Failed Login Redirect to Page
Page

After Logout Redirect to Page
Page

Disable Redirect
☐ Disable

Finally, we enable the checkbox *Use First Supported Mode from Selection* and save our changes of the new content element.

At this point, we are able to login as one of the frontend users.

A typical requirement is to check, if a frontend user is currently logged-in and if not, to redirect the website visitor to the login form as soon as he/she tries to access the Post controller. The `initializeAction()` of the post controller is perfectly suited for this job so let’s edit file `typo3conf/ext/simpleblog/Classes/Domain/Controller/PostController.php` as follows:

```
  ...
```

```
    public function initializeAction() {
        $action = $this->request->getControllerActionName();
        // check, if a different action than "show" was executed
        if ($action != 'show') {
            // redirect to the login page (UID=12), if user is not logged-in
            if (!$GLOBALS['TSFE']->fe_user->user['uid']) {
                $this->redirect(NULL, NULL, NULL, NULL, 12);
            }
        }
    }
    ...
```

The `initializeAction` is executed before every other action so we can use that method to check the status of the current visitor/user. Only the action `show` does not need this kind of check because users not logged in should be able to read Posts of course.

The array `$GLOBALS['TSFE']->fe_user->user['uid']` contains a value, if a frontend user is currently logged-in, otherwise it is not set. By checking this and – in case it is not set – redirecting the user to the login form, we can achieve what we are after.

In our example, the UID is 12 (make sure to set the correct UID of the page with the login form you created) but we recommend not to hard-code the value in the source code. It would be much better to enable integrators to configure the page UID in TypoScript (e.g. `plugin.tx_simpleblog.settings.loginpage = 12`) nand to set this configuration value dynamically in your code (e.g. `$this->redirect(NULL, NULL, NULL, NULL, $this->settings['loginpage']);`.

Let’s remove the author selection from the partial `typo3conf/ext/simpleblog/Resources/Private/Partials/Post/Form.html` again, as well as the transfer of the authors in templates `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/AddForm.html` and `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/UpdateForm.html`.

Then, we should also remove the retrieval of the authors from the `addFormAction()` and `updateFormAction()` in the Post controller `typo3conf/ext/simpleblog/Classes/Domain/Controller/PostController.php`. However we have to adjust `addAction()` as follows:

```
...
    public function addAction(
            \Lobacher\Simpleblog\Domain\Model\Blog $blog,
            \Lobacher\Simpleblog\Domain\Model\Post $post) {
        $post->setPostdate(new \DateTime());
        $post->setAuthor( $this->objectManager->get( 'Lobacher\\Simpleblog\\
        $blog->addPost($post);
...
```

As you can see, we get the UID of the current user by `$GLOBALS['TSFE']->fe_user->user['uid']` and retrieve the author object by executing method `findOneByUid()` of the author repository. This is set via the setter `setAuthor()` of the Post object and the author is assigned correctly.

12.6. Comments & AJAX

We have implemented all domain objects now, which means, the remaining element is the comments. In theory, the functionality to show and process comments could be developed by using the CRUD process again (with a custom controller for example) but from a usability perspective, there is a more efficient method.

It would be better to let users add a comment directly at the single view of a Post. Also, with the use of AJAX, we could store the new comment when a user clicks the *Submit* button and update the list of comments straight away.

The example in this section also shows how to deal with AJAX inside Extbase.

Generally speaking, our requirement should be to implement an AJAX action, which will store the new comment and return the list of existing comments. This should happen entirely in the background and the data returned should not be HTML but JSON.

12.6.1. Registering the AJAX Action

As before, the action has to be registered in file `ext_localconf.php`. Add the name of action `ajax` to the post object:

```
\TYPO3\CMS\Extbase\Utility\ExtensionUtility::configurePlugin(
    ...
    array(
        ...
        'Post' => 'addForm,add,show,updateForm,update,deleteConfirm,delete,a
    ),
    array(
        ...
        'Post' => 'addForm,add,show,updateForm,update,deleteConfirm,delete,a
    )
);
```

12.6.2. Display of Comments

Next we have to expand the show template of the Post controller to display the comments. Edit file `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/Show.html` accordingly:

```
...
<dd>{post.author.fullname} (Email: {post.author.email})</dd>
</dl>

<h3>Comments</h3>
<f:form action="ajax" name="comment" object="{comment}" arguments="{post:pos
    <f:form.textarea property="comment" id="commentfield" />
    <f:form.submit value="Submit comment" class="btn btn-primary btn-xs" id=
</f:form>

<ul class="list-group" id="comments">
<f:for each="{post.comments}" as="comment" reverse="TRUE">
    <li class="list-group-item">{comment.comment} <span class="text-muted">(
</f:for>
</ul>

// JavaScript function

<f:link.action action="show" controller="Blog" arguments="{blog:blog}" class

</f:section>
```

The code above shows a form, which is responsible for the object `comment`. It contains only one field, which has the same name: `comment`. This field will store the comment.

Below the form, the comments are listed inside the `<f:for>` ViewHelper by iterating the property `{post.comments}`. The attribute `reverse` ensures, that the first entry appears at the top of the list, this means that the latest comment is shown first.

12.6.3. JavaScript Handler

Now let’s add the JavaScript function to the same template file `Show.html`, at the position `// JavaScript function`:

```
<script type="text/javascript">
    $(document).ready(function(){
        $('#commentsubmit').click(function(){
            var ajaxURL = '<f:uri.action action="ajax" controller="Post" pag
            var form = $('form');
            $.post(ajaxURL, form.serialize(), function(response) {
                console.log(response);
                var items = [];
                $.each(response, function(key, val) {
                    items.push('<li class="list-group-item">' + val.comment
                });
                $('#comments').html(items.reverse().join(''));
                $('#commentfield').val('');
            });
            return false;
        });
    });
</script>
```

This JavaScript works as follows:

- As soon as the DOM is completely loaded (`$(document).ready(function()`), a click-handler is applied to the submit button with the ID `#commentsubmit`.
- If a user clicks this button, an AJAX function `$.post` is executed. The first parameter of this function is the URL to request, the second contains the serialised form data and the third is a callback function, which will be executed automatically, as soon as the AJAX function returns any data.
- The URL is built by the `<f:uri.action>` ViewHelper and contains the action `ajax`, as well as the current Post: `arguments="{post:post}"`. In addition, attribute `pageType="99"` sets a new page type, which will deliver the JSON response.
- Once the request is successful and the callback function is executed, the response is logged in the console (can be reviewed with the usually build-in developer tools of your browser).
- After that, the JSON data is being parsed and each entry is wrapped in `<li>...</li>` tags.
- Function `reverse` inverts the order of the array so that the newest comment appears at the top of the list again and the content of `<ul id="comments">...</ul>` is replaced by the data.
- Finally, the form’s input field is emptied.

console.log and IE

In Microsoft’s Internet Explorer, the object `console` is not available until the developer tools have been enabled (F12). Otherwise, issues may occur.[36]

12.6.4. AJAX Action in Post controller

Before turning to the AJAX action, we have to correct a minor issue in `initializeAction()`. Edit file `typo3conf/ext/simpleblog/Classes/Controller/PostController.php` as follows:

```
public function initializeAction() {
       $action = $this->request->getControllerActionName();
       if ($action != 'show' && $action != 'ajax') {
       ...
```

We add the action `ajax` to the list of actions, which can be accessed without authentication. You could also extend this example and require users to login before they can submit comments.

Then, create the new action `ajaxAction()` as shown below. We have chosen the name `ajaxAction` for didactic reasons. In production, it might be wiser to choose a name that describes the purpose better, e.g. `addCommentViaAjaxAction()`.

```
    /**
     * @param \Lobacher\Simpleblog\Domain\Model\Post $post
     * @param \Lobacher\Simpleblog\Domain\Model\Comment $comment
     * @return bool|string
     */
    public function ajaxAction(
        \Lobacher\Simpleblog\Domain\Model\Post $post,
        \Lobacher\Simpleblog\Domain\Model\Comment $comment = NULL) {

        // if comment is empty, do not make it persistent
        if ($comment->getComment()=="") return FALSE;

        // set datetime of comment and add comment to Post
        $comment->setCommentdate(new \DateTime());
        $post->addComment($comment);
        $this->postRepository->update($post);
        $this->objectManager->get( 'TYPO3\\CMS\\Extbase\\Persistence\\Generi

        $comments = $post->getComments();
        foreach ($comments as $comment){
            $json[$comment->getUid()] = array(
                    'comment'=>$comment->getComment(),
                    'commentdate' => $comment->getCommentdate()
            );
        }

        return json_encode($json);
    }
```

Firstly, the method has two input parameters: the Post `$post` has been passed as an argument array and the comment `$comment` reflects the content of the form – both set in the `show` action of the Post controller. In the case that the comment is empty, we added =

`NULL`. The first thing we do in the method body is to check if the comment is empty and if so, return `FALSE` and end the action straight away.

If the comment is valid, we set the time stamp of the comment and add it to the Post, which requires to update its repository. After that, we make the record persistent by executing method `persistAll()` of the persistence manager.

Finally, we iterate all comments of the Post (which also includes the new comment now) and build an array. This array contains the UIDs as its keys and sub-arrays with comments and their creation time stamps. In the end the array is converted into a JSON format and returned.

12.6.5. Define AJAX Page Type in TypoScript

Unfortunately, as the output stands now, it contains all data from the TYPO3 framework, which is not usable at this point. We defined a specific page type in the JavaScript code already but we have not configured it yet. So, edit file `typo3conf/ext/simpleblog/Configuration/TypoScript/setup.txt` and add the following to it at the end:

```
ajax = PAGE
ajax {

    typeNum = 99
    config {
        disableAllHeaderCode = 1
        additionalHeaders = Content-type:application/json
        xhtml_cleaning = 0
        admPanel = 0
        debug = 0
    }

    10 < tt_content.list.20.simpleblog_bloglisting
}
```

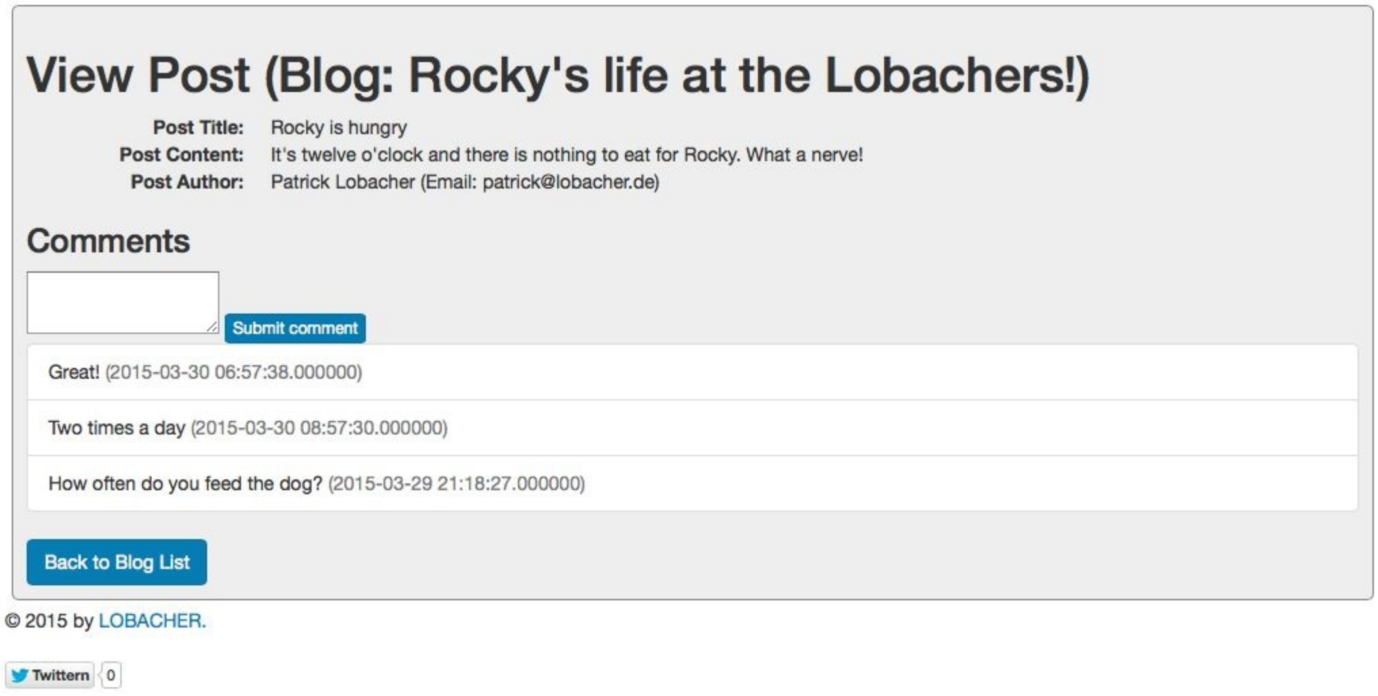
The option `typeNum = 99` defines the appropriate page type. Make sure, this number is not used yet, or choose a different number, possibly with a much higher value. If you do so, do not forget to adjust the number in the JavaScript function, too.

All headers are removed by the configuration `disableAllHeaderCode` and a proper JSON content type is set by additionalHeaders. The remaining settings are default settings and therefore we can copy them by using `10 < tt_content.list.20.simpleblog_bloglisting`.

In the chapter “Best Practices” we will learn how to create this from scratch by using Bootstrap functions, the controller, an action and a few additional parameters.

At this point, our AJAX comment function is ready to use. To test it, access a Post and enter a comment. On submit, the list of comments gets updated and it includes the new entry.

Figure 12.13. Add Comments Via AJAX


View Post (Blog: Rocky's life at the Lobachers!)

Post Title: Rocky is hungry
Post Content: It's twelve o'clock and there is nothing to eat for Rocky. What a nerve!
Post Author: Patrick Lobacher (Email: patrick@lobacher.de)

Comments

Great! (2015-03-30 06:57:38.000000)

Two times a day (2015-03-30 08:57:30.000000)

How often do you feed the dog? (2015-03-29 21:18:27.000000)

© 2015 by LOBACHER.

Commenting for logged-in users only

As an exercise, you could re-build the commenting function so that only authenticated users may post comments. The mechanism is similar to the one we used for the authors but with the difference, that a user registration is also needed because we do not know the authors comments in advance. The extension `femanager`[37] could be used for this purpose. Another option is to ask visitors for their name and email address when they enter a comment. In this case, the domain model and form needs to be adjusted accordingly. If you want to restrict comments to logged-in users, a ViewHelper can be useful, which shows the form only, if the user is authenticated. If not, a link to a login form is shown instead:

```
<f:security.ifAuthenticated>
    <f:then>
        User is logged in!
    </f:then>
    <f:else>
        User is *not* logged in!
    </f:else>
</f:security.ifAuthenticated>
```

However this is “security by obscurity” because the action is executable anyway. An additional check in the action should be taken for granted.

You might wonder why we reload all the comments and not just the one we submitted? The reason for this is that the form can be accessed and another comment submitted by someone else at the same time. If we only retrieved and displayed our own entry, we would not see other comments until the page is reloaded.

---

[34] http://php.net/manual/en/class.splobjectstorage.php

[35] http://docs.typo3.org/typo3cms/TCAReference/

[36] Further details and workarounds are available at: http://stackoverflow.com/questions/690251/what-happened-to-console-log-in-ie8

[37] http://typo3.org/extensions/repository/view/femanager

Chapter 13. Creating Your Own ViewHelpers

ViewHelpers are PHP classes, which support the view logic and are used in Fluid. They usually come into play when Fluid's standard set of functions are not sufficient for a specific task.

Currently, Fluid is shipped with approximately 100 ViewHelpers (they are stored in directory `typo3/sysext/fluid/Classes/ViewHelpers/`) but you can also create your own.

There are usually four types of ViewHelpers:

Text-ViewHelper
: Generate any kind of texts, e. g. markup.

Tag-ViewHelper
: Render a HTML tag and outputs this.

If-ViewHelper
: Make decisions, based on conditions and branch into either “in-this-case” (then) or “is-not-the-case” (else).

Widget-ViewHelper
: Have their own controller and their own view and are used predominately when an additional control structure is required.

13.1. Namespace Declaration

In order to use a ViewHelper (except all built-in ViewHelpers), a namespace declaration needs to be added to every template (layout, partial). This declaration contains the acronym of the namespace (e.g. `f` for build-in ViewHelpers) and the directory, where the PHP class is stored.

The namespace declaration is stated as follows:

```
{namespace abbreviation = file system path}
```

For example:

```
{namespace pl = Lobacher\Simpleblog\ViewHelpers}
```

The number of declarations is not limited but acronyms must be unique.

TYPO3 resolves the path as follows:

- First, TYPO3 checks, if a directory simpleblog exists under `typo3conf/ext/`
- If it does not exist, directory `typo3/sysext/` is checked
- Then, directory `Classes/ViewHelpers/` is used as a reference
- If a ViewHelper with the appropriate name is used, Extbase searches for a file `[ViewHelperName]ViewHelper.php` in this directory
- If the ViewHelper contains a dot (e.g. `format.html`), the last element always builds the name of the ViewHelper (in this case `HtmlViewHelper.php`) and all elements before (an arbitrary number of elements) are sub-directories.

13.2. Text ViewHelper

Text ViewHelpers return any kind of text, which could be markup texts such as HTML or XML too. This type is always derived from the abstract class `\TYPO3\CMS\Fluid\Core\ViewHelper\AbstractViewHelper`.

For our Blog, we will implement a ViewHelper, which retrieves a value from the TSFE (TypoScript Frontend). This can be used to output the page title or the last name of the currently logged-in user for example.

Create a new file `typo3conf/ext/simpleblog/Classes/ViewHelpers/TsfeViewHelper.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers;

class TsfeViewHelper extends \TYPO3\CMS\Fluid\Core\ViewHelper\AbstractViewHe

    /**
     * @param $key string
     * @return string
     */
    public function render($key) {
        if ($key === NULL) {
            return '';
        } else {
            return $this->getTsfeValue('TSFE|'.$key);
        }
    }
    public function getTsfeValue($keyString) {
        $keys = explode('|', $keyString);
        $numberOfLevels = count($keys);
        $rootKey = trim($keys[0]);
        $value = $GLOBALS[$rootKey];
        for ($i = 1; $i < $numberOfLevels && isset($value); $i++) {
            $currentKey = trim($keys[$i]);
            if (is_object($value)) {
                $value = $value->{$currentKey};
            } elseif (is_array($value)) {
                $value = $value[$currentKey];
            } else {
                $value = '';
                break;
            }
        }
        if (!is_scalar($value)) {
            $value = '';
        }
        return $value;
    }
}
```

Every ViewHelper must feature a method `render()`, that returns the content. As the input parameters, either specific arguments (in our example: `$key`) or the content between the opening and closing tag can be used.

13.2.1. Parameter Via Attribute

We will use the method via an argument and place the ViewHelper into the template `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/List.html` as shown below.

```
{namespace pl = Lobacher\Simpleblog\ViewHelpers}
<f:layout name="default" />
...
<h1>Blog List</h1>
<h3><pl:tsfe key="page|subtitle" /></h3>
...
```

Via parameter `$key` in method `render()`, we can access the values of the attribute `key="page|subtitle"` of the ViewHelper and process them. It is important that attribute name and method parameter have the same names.

This works perfectly fine and is intuitive, as long as you do not have too many attributes. If the number grows, you should consider using the dedicated method `registerArgument()`, which is available in method `initializeArguments()`:

```
...
class TsfeViewHelper extends \TYPO3\CMS\Fluid\Core\ViewHelper\AbstractViewHe

    public function initializeArguments() {
        $this->registerArgument('key', 'string',
            'This is the TSFE key e.g. page|title', TRUE);
    }

    /**
     * @return string
     */
    public function render() {
        if ($this->arguments['key'] === NULL) {
            return '';
        } else {
            return $this->getTsfeValue('TSFE|'.$this->arguments['key']);
        }
    }
    ...
```

Method `registerArgument` is defined as follows:

```
protected function registerArgument($name, $type, $description, $required =
```

`$name`
: Contains the name of the argument, which is to be registered.

`$type`
: Contains the type (e.g. `string`, `boolen`, etc.).

`$description`
: Allows us to store a description.

`$required`
: `FALSE` (this is the default), which means the agument is optional. If you want to make it mandatory, set this parameter to `TRUE`.

`$defaultValue`
: Sets a default value, which will be used, if the argument is empty or does not exist.

13.2.2. Parameter Via Content

There is an additional method of how to access the input for a ViewHelper. Everything between the opening and closing ViewHelper tag can be used as the input data.

The Method `$this->renderChildren()` takes care of this by rendering the data, even if they contain further ViewHelpers.

```
public function initializeArguments() {
    $this->registerArgument('key', 'string',
        'This is the TSFE key e.g. page|title', FALSE);
}

/**
 * @return string
 */
public function render() {
    $key = ($this->arguments['key']) ? $this->arguments['key'] : $this->
    if ($key === NULL) {
        return '';
    } else {
        return $this->getTsfeValue('TSFE|'.$key);
    }
}
```

The parameter required must be set to `FALSE` at the registration. Then, inside method `render()` it is checked, if an argument key has been passed. If not, the data between the tags is read by the function call `$this->renderChildren()`.

The implementation of the ViewHelper in the view looks like this:

```
<h3><pl:tsfe>page|title</pl:tsfe></h3>
```

The content `page|title` could also come from another ViewHelper now.

13.3. Tag ViewHelper

The Tag ViewHelper outputs XML and therefore extends the text ViewHelper described before.

It derives from the abstract class `\TYPO3\CMS\Fluid\Core\ViewHelper\AbstractTagBasedViewHelper`.

In order to demonstrate this ViewHelper, we will show a “Gravatar” – a specific image based on an email address.[38]

Create a new file `typo3conf/ext/simpeblog/Classes/ViewHelpers/GravatarViewHelper.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers;

class GravatarViewHelper extends \TYPO3\CMS\Fluid\Core\ViewHelper\AbstractTa

    protected $tagName = 'img';

    public function initializeArguments() {
        $this->registerArgument('email', 'string',
            'Email for lookup at gravatar database', FALSE);
        $this->registerArgument('size', 'integer',
            'Size of gravatar picture', FALSE, 100);
    }

    /**
     * @return string the HTML <img>-Tag of the gravatar
     */
    public function render() {
        $email = ($this->arguments['email'] !== NULL) ? $this->arguments['em

        $gravatarUri = 'http://www.gravatar.com/avatar/' . md5($email) . '?s

        $this->tag->addAttribute('src', $gravatarUri);
        return $this->tag->render();
    }

}
```

The protected variable `$tagName` specifies the HTML tag, which will be used later, when the output is rendered in `$this->tag->render()`. Without this variable, the ViewHelper would return a `<div>`-tag.

After that, two arguments are registered: `email` (the email address) and `size` (the size of the Gravatar in pixels, default value is 100px).

Inside the obligatorily method `render()`, argument email is read. If this argument does not exist, the ViewHelper tries to find an email address by `$this->renderChildren()`.

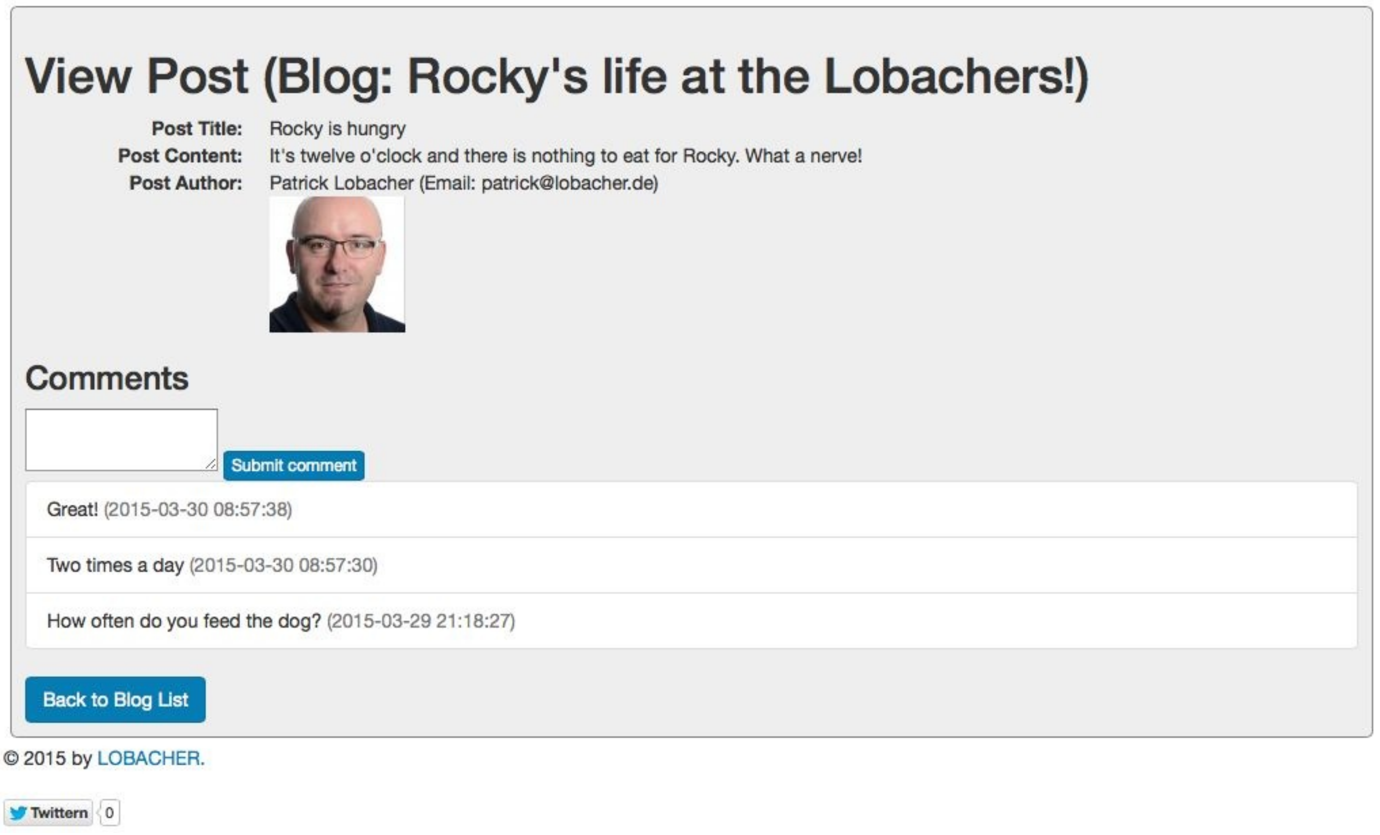
Next, the URL is built by appending the MD5 hash of the email address to the URL of the Gravatar service. Parameter `?s=` defines the size of the image.

Finally, the URL is added as the `src` attribute by the `$this->tag->addAttribute()` call and the tag is rendered and returned.

The ViewHelper is now ready to be used in template `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/Show.html` to display the Gravatar as an image:

```
{namespace pl = Lobacher\Simpleblog\ViewHelpers}
<f:layout name="default" />
...
    <dt>Post Author:</dt>
    <dd>{post.author.fullname} (email: {post.author.email})
    <br />
    <pl:gravatar>{post.author.email}</pl:gravatar></dd>
</dl>
```

Figure 13.1. Custom ViewHelper shows Gravatar



View Post (Blog: Rocky's life at the Lobachers!)

Post Title:	Rocky is hungry
Post Content:	It's twelve o'clock and there is nothing to eat for Rocky. What a nerve!
Post Author:	Patrick Lobacher (Email: patrick@lobacher.de)

Comments

Great! (2015-03-30 08:57:38)

Two times a day (2015-03-30 08:57:30)

How often do you feed the dog? (2015-03-29 21:18:27)

© 2015 by LOBACHER.

To register attributes such as `class`, `dir`, `id`, `lang`, `style`, `title`, `accesskey`, `tabindex` or `onclick` method `$this->registerUniversalTagAttributes();` provides a nice shortcut.

Calling this function method `initializeArguments()` registers the attributes listed above.

13.4. If ViewHelper

As pointed out before, the If ViewHelper checks a condition and branches into either “in-this-case” (then) or “is-not-the-case” (else).

ViewHelpers of this type always derive from abstract class `\TYPO3\CMS\Fluid\Core\ViewHelper\AbstractConditionViewHelper`.

The example ViewHelper in this section checks, if we are in the frontend context. If this is the case, the Then-ViewHelper is executed, otherwise the Else-ViewHelper. This ViewHelper will become very important when we develop a backend module, which uses the same Fluid code as our frontend extension. Some specific functions should only be visible in the frontend, others only in the backend.

For our new ViewHelper, a new file `typo3conf/ext/simpeblog/Classes/ViewHelpers/IsFrontendViewHelper.php` is required:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers;

class IsFrontendViewHelper extends \TYPO3\CMS\Fluid\Core\ViewHelper\Abstract

    public function render() {
        if (TYPO3_MODE === 'FE') {
            return $this->renderThenChild();
        }
        return $this->renderElseChild();
    }

}
```

We can insert the new ViewHelper into a view as the following example shows:

```
{namespace pl = Lobacher\Simpleblog\ViewHelpers}
...
<pl:isFrontend>
    <f:then>
        Frontend!
    </f:then>
    <f:else>
        Backend!
    </f:else>
</pl:isFrontend>
...
```

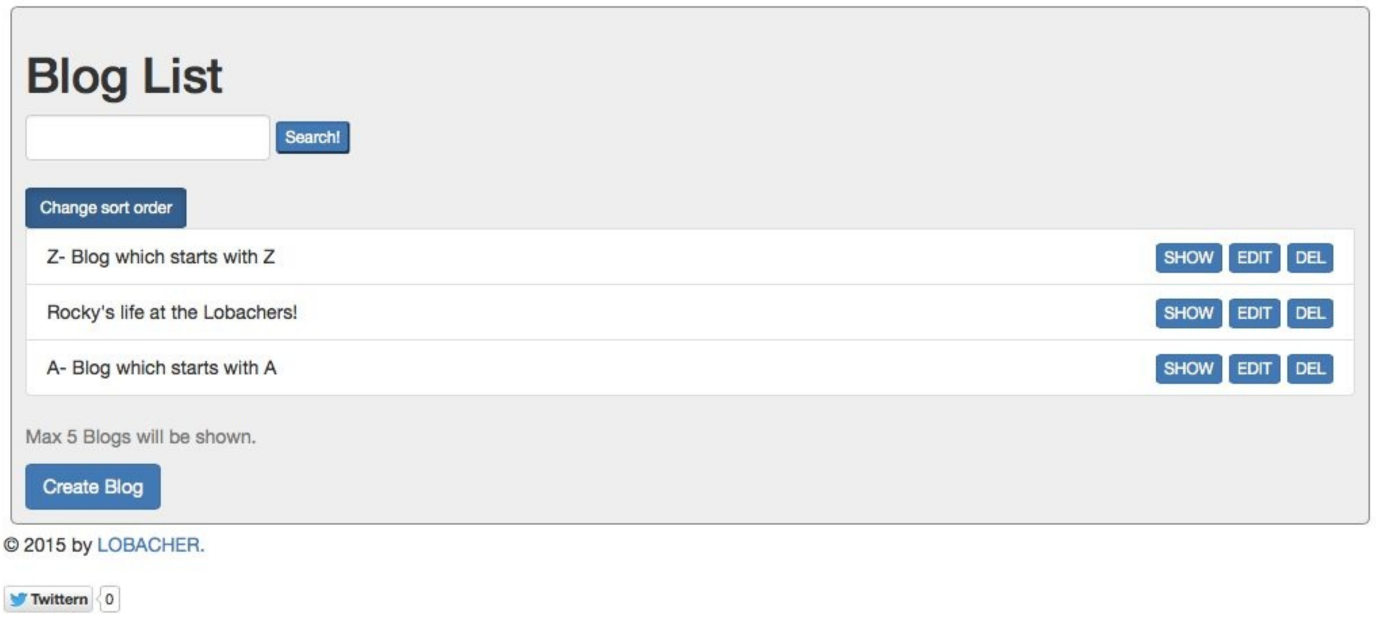
13.5. Widget ViewHelper

Widgets are ViewHelper, which feature their own controller and view. They are usually used where additional and specific control of the view is required, for example to implement a page browser, a sorting function, an auto-completion or similar.

Let’s have a look at a typical sorting functionality of a Blog listing:

- The view shows a list of Blogs.
- A link with a specific parameter allows users to change the sort order.
- Blog controller accepts this parameter in the list action and passes it on to the repository.
- The repository changes the sort order accordingly and returns the updated list of Blogs back to the controller.
- The new list gets passed to the view, where it is displayed.

Figure 13.2. ViewHelper with sorting feature


Blog List
Search!
Change sort order
Z- Blog which starts with Z
SHOW
EDIT
DEL
Rocky's life at the Lobachers!
SHOW
EDIT
DEL
A- Blog which starts with A
SHOW
EDIT
DEL
Max 5 Blogs will be shown.
Create Blog
© 2015 by LOBACHER.
Twittern
0

This does not seem too complex but does have a significant drawback: the controller is integrated in the view logic. However Extbase strictly separates the domain logic from the view logic, which means, the controller should not participate in this process at all.

For this reason Widgets have been introduced in TYPO3 version 4.5 LTS (Extbase version 1.3) and the concept of repositories revised. Previously, repositories returned data sets directly. This has been changed in the way that repositories merely return an interface (more precisely: `\TYPO3\CMS\Extbase\Persistence\QueryInterface`) now, which allow Widgets to manipulate it in the view. Not before the view outputs the data, Extbase accesses the repository. Widgets always remain ViewHelpers in this case – if only special ones.

In order to influence/control the data, a controller, actions and a view are required and this is in the context of the view. Therefore this is called *subrequest*.

The following sections explain how such a sorting Widget could be developed. The requirements are to implement a link at the Blog Posts page, which changes the sorting (by `title`) if clicked, ascending and vice versa.

13.5.1. Use of Widget ViewHelpers

Let’s begin with the list view of Blogs:

```
typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/List.html

{namespace pl = Lobacher\Simpleblog\ViewHelpers}
...
<ul class="list-group">
    <pl:widget.sort objects="{blogs}" as="sortedBlogs" property="title">

    <f:for each="{sortedBlogs}" as="blog">
        <li class="list-group-item">{blog.title}
            ...
        </li>
    </f:for>
    </pl:widget.sort>
</ul>
...
```

As you can see, we have added a Widget (more precisely: a ViewHelper) named `widget.sort`. The attribute `objects` passes all Blogs to the Widget. The (re-ordered) list gets returned as `sortedBlogs` and the `property` of the object, by which the sorting should happen, is defined as a title.

13.5.2. Creation of Widget ViewHelpers

Continue with the ViewHelper itself: first, create a new directory `typo3conf/ext/simpleblog/Classes/ViewHelpers/Widget` and inside this, a new file `SortViewHelper.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers\Widget;

class SortViewHelper extends \TYPO3\CMS\Fluid\Core\Widget\AbstractWidgetView

    /**
     * @var \Lobacher\Simpleblog\ViewHelpers\Widget\Controller\SortControlle
     * @inject
     */
    protected $controller;


    /**
     * @param \TYPO3\CMS\Extbase\Persistence\QueryResultInterface $objects
     * @param string $as
     * @param string $property
     * @return string
     */
    public function render(\TYPO3\CMS\Extbase\Persistence\QueryResultInterfa
        return $this->initiateSubRequest();
    }
}
```

Widget ViewHelpers always derive from abstract class `\TYPO3\CMS\Fluid\Core\Widget\AbstractWidgetViewHelper`.

After that, we fetch the appropriate controller by using Dependency Injection so it becomes available in our class.

Same as before, a method `render()` exists, which receives the objects via attribute `objects`, the name of the result set via attribute `as` and the object properties via attribute `property`.

Finally, method call `$this->initiateSubRequest()` triggers the subrequest, which takes care of executing the action `indexAction()` of the controller, which has been fetched by DI.

13.5.3. The Controller

To implement the controller, we start with creating a new directory `typo3conf/ext/simpleblog/ViewHelpers/Widget/Controller` and inside that, a new file `SortController.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers\Widget\Controller;

class SortController extends \TYPO3\CMS\Fluid\Core\Widget\AbstractWidgetCont

    /**
     * @var \TYPO3\CMS\Extbase\Persistence\QueryResultInterface
     */
    protected $objects;

    public function initializeAction() {
        $this->objects = $this->widgetConfiguration['objects'];
    }

    /**
     * @param string $order
     */
    public function indexAction($order = \TYPO3\CMS\Extbase\Persistence\Quer
        $order = ($order == \TYPO3\CMS\Extbase\Persistence\QueryInterface::O

        $query = $this->objects->getQuery();
        $query->setOrderings(array($this->widgetConfiguration['property'] =>
        $modifiedObjects = $query->execute();

        $this->view->assign('contentArguments', array(
            $this->widgetConfiguration['as'] => $modifiedObjects
        ));
        $this->view->assign('order', $order);
    }

}
```

The controller of the Widget always derive from abstract class `\TYPO3\CMS\Fluid\Core\Widget\AbstractWidgetController`.

In method `initializeAction()`, we access the objects, which have been passed through the attribute `objects` by using `$this->widgetConfiguration['...']`.

The only method parameter of the index action is `$order`, which is passed as an action link (we will see this in more detail in a minute). Directly after that, we check what the current setting of `$order` is. In the case it is `ASC` (ascending), we set it to `DESC` (descending) and vice versa. For the sorting, official constants are used (`ORDER_ASCENDING` or `ORDER_DESCENDING`).

Due to the fact that the Query-Interface is available as `$objects`, we can access it directly. The query is fetched via `getQuery()` and by calling `setOrdering()`, the sort order is applied. `$this->widgetConfiguration['property']` helps us to determine by which property the data should be sorted and method `execute()` triggers the query.

The re-sorted objects are passed as `contentArguments` to the view, as well as the current (new) sort order as `order`.

13.5.4. The View

The view exists at the location, where all other templates can be found but in a sub-directory `ViewHelpers/Widget/Sort`. Let’s create the following directories:

```
typo3conf/ext/simpleblog/Resources/Private/Templates/ViewHelpers
typo3conf/ext/simpleblog/Resources/Private/Templates/ViewHelpers/Widget
typo3conf/ext/simpleblog/Resources/Private/Templates/ViewHelpers/Widget/Sort
```

Inside the last directory, we create a new file `Index.html` with the following content:

```
<f:widget.link arguments="{order:order}" class="btn btn-primary btn-sm activ

<f:renderChildren arguments="{contentArguments}" />
```

The view contains some specific ViewHelpers, which are often useful and help us even further:

`<f:widget.link>`
: This ViewHelper creates a specific Widget action link and has parameters action (the action, default is `index`), `arguments` (allows passing arguments), `section` (manipulates the URL if required) and `format` (specifies the format, default is `.html`). Another parameter `ajax` specifies, if the Link leads to an AJAX Widget or not (default is `FALSE`).

`<f:widget.uri>`
: This Widget is identical to `widget.link` but returns the URL instead of the HTML tag.

`<f:renderChildren>`
: makes sure that the child node (the content of the ViewHelper) is being rendered.

13.5.5. The A to Z Widget

Now you should try to implement your own Widget: a navigation bar that shows letters A to Z and if a user clicks one of the letters, only Blogs should come up where their title starts with this letter. Let’s choose `widget.AtoZNav` as the name of the Widget. It’s best not to continue reading this chapter, until you have successfully finished this task or you really struggle to implement this feature.

Start with the view: update the list view of the Blog by editing the file `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/List.html`:

```
<ul class="list-group">
    <pl:widget.AtoZNav objects="{blogs}" as="filteredBlogs" property="title"
        <pl:widget.sort objects="{filteredBlogs}" as="sortedBlogs" property=
            <f:for each="{sortedBlogs}" as="blog">
                ...
            </f:for>
        </pl:widget.sort>
    </pl:widget.atoZNav>
</ul>
```

Continue with the file `typo3conf/ext/simpleblog/Classes/ViewHelpers/Widget/AtoZNavViewHelper.php` and add the following content:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers\Widget;

class AtoZNavViewHelper extends \TYPO3\CMS\Fluid\Core\Widget\AbstractWidgetV

    /**
     * @var \Lobacher\Simpleblog\ViewHelpers\Widget\Controller\AtoZNavContro
     * @inject
     */
    protected $controller;

    /**
     * @param \TYPO3\CMS\Extbase\Persistence\QueryResultInterface $objects
     * @param string $as
     * @param string $property
     * @return string
     */
    public function render(\TYPO3\CMS\Extbase\Persistence\QueryResultInterfa
        return $this->initiateSubRequest();
    }
}
```

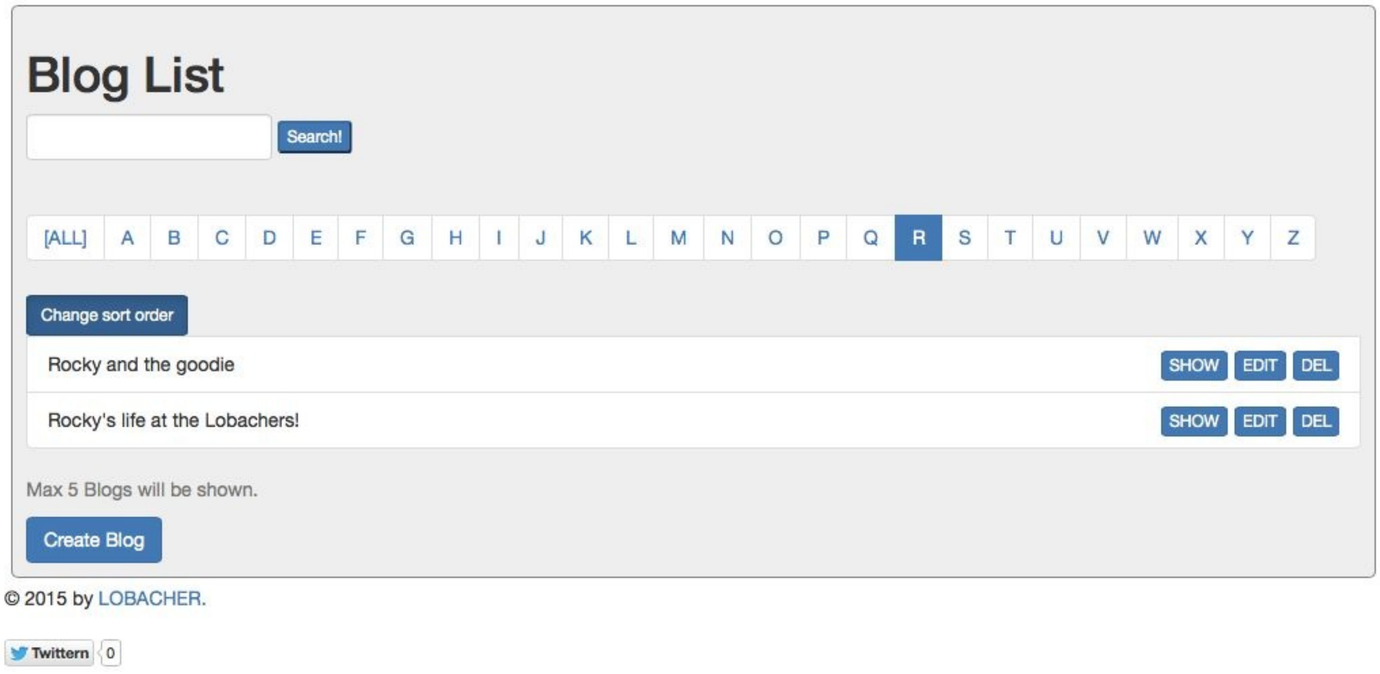
Followed by implementing the controller `typo3conf/ext/simpleblog/Classes/ViewHelpers/Widget/Controller/AtoZNavContro` with the following content:

```
<?php
namespace Lobacher\Simpleblog\ViewHelpers\Widget\Controller;

class AtoZNavController extends \TYPO3\CMS\Fluid\Core\Widget\AbstractWidgetC

    /**
     * @var \TYPO3\CMS\Extbase\Persistence\QueryResultInterface
     */
    protected $objects;

    public function initializeAction() {
        $this->objects = $this->widgetConfiguration['objects'];
    }

    /**
     * @param string $order
     */
    public function indexAction($char = '%') {

        $query = $this->objects->getQuery();
        // get selected objects only (title starting with specific letter)
        $query->matching($query->like($this->widgetConfiguration['property']
        $modifiedObjects = $query->execute();

        $this->view->assign('contentArguments', array(
            $this->widgetConfiguration['as'] => $modifiedObjects
        ));

        // create an array with all letters from A to Z
        foreach (range('A', 'Z') as $letter) {
            $letters[] = $letter;
        }
        $this->view->assign('letters', $letters);
        $this->view->assign('char', $char);
    }

}
```

Now create the View as file `typo3conf/ext/simpleblog/Resources/Private/Templates/ViewHelpers/Widget/AtoZ` and the following content:

```
<ul class="pagination">
    <li{f:if(condition:'{char}=="%"',then:' class="active"')}><f:widget.link
    <f:for each="{letters}" as="letter">
        <li{f:if(condition:'{char}=={letter}',then:' class="active"')}>
            <f:widget.link arguments="{char:letter}" class="active">{letter}
        </li>
    </f:for>
</ul>
<br />

<f:renderChildren arguments="{contentArguments}" />
```

Figure 13.3. A to Z list



Rocky and the goodie

Rocky's life at the Lobachers!

Max 5 Blogs will be shown.

© 2015 by LOBACHER.

---

[38] This requires a (free of charge) registration at http://www.gravatar.com and an image uploaded to your email address. If you use your email address for other services, which also support Gravatars (e.g. WordPress Blogs), the address is passed MD5 encoded to gravatar.com, which returns the uploaded image, or (if no image can be found) a dummy image.

Chapter 14. Multi-Language

For didactic reasons, we have overlooked the language so far. We will address this now and introduce multi-language features in all areas of our project.

14.1. Language Configuration

There are a substantial number of configuration options in TYPO3, which are all well documented and can be found in the “Frontend Localization Guide”[39]

A frontend language is basically created in the *List* module on page UID `0` (data type: *Website language*), except the default language. The latter has UID `0` and it is not required to create it explicitly. Each further language features the UID of the data set.

After that, the language handling must be configured in TypoScript. Therefore, add the following code to the setup of your TypoScript template:

```
config.linkVars = L
config.uniqueLinkVars = 1

config {
   sys_language_uid = 0
   language = default
   locale_all = en_GB
   htmlTag_langKey = en
}

[globalVar = GP:L = 1]
config {
   sys_language_uid = 1
   language = de
   locale_all = de_DE.utf8
   htmlTag_langKey = de
}
[global]
```

This has the following impact:

- `linkVars` ensures, that the language parameter `L` is added to all internally generated links.
- The default language is configured to be English.
- If a request with the language parameter `1` hits the TYPO3 instance (e.g. by `&L=1` in the URL), the TypoScript condition `[globalVar = GP:L = 1]` re-configures the language handling to be German (`de`).

14.2. Language Labels

In order to make labels multi-lingual, a *localisation file* is required. This is a XML file, which follows a specific syntax. In TYPO3 version 4.6 and before, a propriety format was used (usually `locallang.php` and later `locallang.xml`). Today, the modern XLIFF format (*XML Localization Interchange Format*) comes into play.

Extbase expects this file under
`typo3conf/ext/simpleblog/Resources/Private/Language/locallang.xlf`.

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<xliff version="1.0">
    <file source-language="en" datatype="plaintext" original="messages" date
        <header/>
        <body>
...
            <trans-unit id="headline.blog">
                <source>Blog List</source>
            </trans-unit>
...
        </body>
    </file>
</xliff>
```

Add a section `<trans-unit>` with ID `headline.blog` to the code shown above (see example).

In order to create a translation, create a copy and rename the file to `de.locallang.xlf`. As you can see, the two-characters ISO code is put in front.

The content of the file should be amended as follows:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<xliff version="1.0">
    <file source-language="en" target-language="de" datatype="plaintext" ori
        <header/>
        <body>
...
            <trans-unit id="headline.blog">
                <source>Blog-List</source>
                <target state="translated">Blog-Liste</target>
            </trans-unit>
...
        </body>
    </file>
</xliff>
```

The core API documentation[40] describes the format of XLIFF files in detail and which other features XLIFF offers.

At this point, we can use the label in the list view of the Blog (file: `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/List.html`) instead of `<h1>Blog List</h1>`.

```
...
<h1><f:translate id="headline.blog" /></h1>
...
```

In order to achieve a translation of the button *Search*, the approach is a little bit different. However the syntax of the language labels remains identical. Let’s start with file `locallang.xlf`:

```
...
            <trans-unit id="button.search">
                <source>Search!</source>
            </trans-unit>
...
```

Then, file `de.locallang.xlf`:

```
...
            <trans-unit id="button.search">
                <source>Search!</source>
                <target state="translated">Suchen!</target>
            </trans-unit>
...
```

The usage of the label happens in the list template of the Blog as well but if we look at the button more closely, we realise that we can not add the translate ViewHelper.

```
...
<f:form.submit value="<f:translate id="button.search" />" class="btn-xs btn-
...
```

The above would result in a syntax error so we have to use the inline syntax:

```
...
<f:form.submit value="{f:translate(id:'button.search')}" class="btn-xs btn-p
...
```

14.3. Language Labels with Placeholders

Looking at the Post reveals that it shows a more complicated headline. It consists not only of the label but also the name of the Blog, which means, we have to use placeholders in this case.

Let’s open the show template of the Post (file: `typo3conf/ext/simpleblog/Resources/Private/Templates/Post/Show.html`) and update the following line:

```
<h1>Show Post (Blog: {blog.title})</h1>
```

The placeholder shown below replaces the hard-coded text:

```
<h1><f:translate key="headline.post" arguments="{1:blog.title,2:blog.uid}" /
```

By using the `arguments` array, we can pass arbitrary data to the language file.

Here, the data gets parsed with a similar logic than the PHP command `sprintf`[41]. Let’s get started – first, file `locallang.xlf`:

```
...
            <trans-unit id="headline.post">
                <source>View post (Blog: %1$s / UID: %2$s)</source>
            </trans-unit>
...
```

Then, file `de.locallang.xlf`:

```
...
            <trans-unit id="headline.post">
                <source>View post (Blog: %1$s / UID: %2$s)</source>
                <target state="translated">Post ansehen (UID: %2$s / Blog: %
            </trans-unit>
...
```

Each placeholder starts with a `%` character, followed by a number which represents the position in the `arguments` array (the first position equates to `1`, the second to `2`, independently from the key, which is ignored). Then, the `$` character and the `sprintf`-code for the formatting, e.g. `s` for a string or `d` for a decimal number.

14.4. Overwrite Language Labels by TypoScript

The key `_LOCAL_LANG.[ISO]` allows TYPO3 integrators to overwrite every language label via TypoScript. In this connection, `[ISO]` reflects the two-characters language label.

To overwrite the headline of the Blog list in German, the following TypoScript can be used:

```
plugin.tx_simpleblog {
   _LOCAL_LANG {
      de {
         headline.blog = Liste aller Blogs
      }
   }
}
```

14.5. Language Labels in PHP

It is also possible to translate language labels in PHP. Function `translate()` serves exactly this purpose:

```
\TYPO3\CMS\Extbase\Utility\LocalizationUtility::translate($key, $extension,
```

The parameters have the following meaning:

`$key`
: Corresponds to the key in the language file, e.g. `headline.blog`.

`$extension`
: The extension, of which file `locallang.xlf` should be used. In the case this should be the file of your own extension, use value `NULL` as this parameter.

`$arguments`
: Optional arguments to pass to the language file as an array. If the language label does not require any arguments, leave the parameters empty or set it to `NULL`.

For example:

```
\TYPO3\CMS\Extbase\Utility\LocalizationUtility::translate('headline.blog', N
```

14.6. Multi-Language for Domain Objects

Extbase handles the multi-language capability with flexibility and transparency, if you take care of the correct fields in the database and their configuration in the TCA.

The database must contain the following fields for every multi-language domain object:

```
...
    sys_language_uid int(11) DEFAULT '0' NOT NULL,
    l10n_parent int(11) DEFAULT '0' NOT NULL,
    l10n_diffsource mediumblob,
...
```

The TCA requires the following definitions (in our example the Blog object):

```
$TCA['tx_simpleblog_domain_model_blog'] = array(
    'ctrl' => array(
       ...
       'languageField' => 'sys_language_uid',
       'transOrigPointerField' => 'l10n_parent',
       'transOrigDiffSourceField' => 'l10n_diffsource',
...
```

Also in the TCA, the fields must be configured properly:

```
$TCA['tx_simpleblog_domain_model_blog'] = array(
    'ctrl' => $TCA['tx_simpleblog_domain_model_blog']['ctrl'],
    ...
    'columns' => array(
        'sys_language_uid' => array(
            'exclude' => 1,
            'label' => 'LLL:EXT:lang/locallang_general.xlf:LGL.language',
            'config' => array(
                'type' => 'select',
                'foreign_table' => 'sys_language',
                'foreign_table_where' => 'ORDER BY sys_language.title',
                'items' => array(
                    array('LLL:EXT:lang/locallang_general.xlf:LGL.allLanguag
                    array('LLL:EXT:lang/locallang_general.xlf:LGL.default_va
                ),
            ),
        ),
        'l10n_parent' => array(
            'displayCond' => 'FIELD:sys_language_uid:>:0',
            'exclude' => 1,
            'label' => 'LLL:EXT:lang/locallang_general.xlf:LGL.l18n_parent',
            'config' => array(
                'type' => 'select',
                'items' => array(
                    array('', 0),
                ),
                'foreign_table' => 'tx_simpleblog_domain_model_blog',
                'foreign_table_where' => 'AND tx_simpleblog_domain_model_blo
            ),
        ),
        'l10n_diffsource' => array(
            'config' => array(
                'type' => 'passthrough',
            ),
        ),
...
```

It is important, that the language is set in the repository accordingly (for example UID `1`). However you should not hard-code the UID of course but use `$GLOBALS['TSFE']->sys_language_uid`:

```
...
$query = $this->createQuery();
...
$query->getQuerySettings()->setSysLanguageUid(1);
...
return $query->execute();
```

Provided that translated records exist in the database, those are returned now. The example evaluates `sys_language_uid = 1`, which means records of the first created language. In the case, that no translations exist, the original (default) records are returned.

Manual configurations are also possible. The example below could be implemented in the controller: a Blog is created, title set, language set to UID `1` and the Blog persisted at the end.

```
$new = $this->objectManager->create( 'Lobacher\\Simpleblog\\Domain\\Model\\B
$new->setTitle('Blog-Titel');
$new->setSysLanguageUid(1);
$new->_setProperty('_languageUid', 1);
$new->setL10nParent(123);
$this->blogRepository->add($new);
$this->objectManager->get( 'TYPO3\\CMS\\Extbase\\Persistence\\Generic\\Persi
```

Needless to say, that the setter for `sys_language_uid` and `l10n_parent` must exist in the model.

```
/**
 * Set sys language
 *
 * @param int $sysLanguageUid
 * @return void
*/
public function setSysLanguageUid($sysLanguageUid) {
    $this->_languageUid = $sysLanguageUid;
}

/**
 * Set l10n parent
 *
 * @param int $l10nParent
 * @return void
*/
public function setL10nParent($l10nParent) {
    $this->l10nParent = $l10nParent;
}
```

---

[39] `doc_guide_l10n` and `doc_l10guide` at http://docs.typo3.org/typo3cms/FrontendLocalizationGuide/

[40] http://docs.typo3.org/typo3cms/CoreApiReference/Internationalization/Index.html

[41] http://php.net/sprintf

Chapter 15. Backend Modules

The backend and frontend modules of Extbase are practically identical. This allows developers to use the same code for the frontend as well as for the backend. As we know, this was not possible under `pi_base` extensions.

In order to demonstrate this capability, we will create a backend module, which enables backend users to delete Blog comments. In addition, entries which were deleted should also be listed.

15.1. Registering the Module

In order to register the module, edit file `ext_tables.php` and call method `registerModule()`, which features the following API:

```
/**
     * Registers an Extbase module (main or sub) to the backend interface.
     * FOR USE IN ext_tables.php FILES
     *
     * @param string $extensionName The extension name (in UpperCamelCase) o
     * @param string $mainModuleName The main module key. So $main would be
     * @param string $subModuleName The submodule key.
     * @param string $position This can be used to set the position of the $
     * @param array $controllerActions is an array of allowed combinations o
     * @param array $moduleConfiguration The configuration options of the mo
     * @throws \InvalidArgumentException
     * @return void
     */
    static public function registerModule(
        $extensionName,
        $mainModuleName = '',
        $subModuleName = '',
        $position = '',
        array $controllerActions,
        array $moduleConfiguration = array()
        ) {
```

In our case, we should consider the following:

`$extensionName`
: Vendor namespace including extension name.

`$mainModuleName`
: Our module should appear under the main module `system`, which is the section where `BackendUser`, `Install`, `Log` etc. are listed.

`$subModuleName`
: Name of our sub module – let's use `SimpleblogAdmin`.

`$position`
: The position of the menu entry in the section. The syntax is `[cmd]:[submodule-key]`, where `[cmd]` can be one of the key words `after`, `before` or `top`. In order to position our module at the top of the section `system`, we choose `top`.

`array $controllerActions`
: Lists the controller/action combinations, which can be executed by the module. The following three are required for our module: `list` (to list the comments), `delete` (to delete a comment) and `test` (for the reader to test).

`array $moduleConfiguration`
: These are additional module configurations: `access` (to limit the access to the module `admin`), `icon` (path to the icon of the module) and `labels` (refers to a language file to

show labels).

Which leads to this method call:

```
if (TYPO3_MODE === 'BE') {
    \TYPO3\CMS\Extbase\Utility\ExtensionUtility::registerModule(
        'Lobacher.' . $_EXTKEY,
        'system',
        'SimpleblogAdmin',
        'top',
        array(
            'Comment' => 'list,delete,test'
        ),
        array(
            'access'    => 'admin',
            'icon'      => 'EXT:' . $_EXTKEY . '/ext_icon.gif',
            'labels'    => 'LLL:EXT:' . $_EXTKEY . '/Resources/Private/Langu
        )
    );
}
```

15.2. Language File for Labels

In the next step, we will create a language file, which has been specified by the option `labels` before. This file should sit under `typo3conf/ext/simpleblog/Resources/Private/Language/locallang_mod.xlf` and contain the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xliff version="1.0">
    <file source-language="en" datatype="plaintext" original="messages" date
        <header/>
        <body>
            <trans-unit id="mlang_labels_tablabel" xml:space="preserve">
                <source>SB: Comment Admin</source>
            </trans-unit>
            <trans-unit id="mlang_labels_tabdescr" xml:space="preserve">
                <source>SB: Comment Admin</source>
            </trans-unit>
            <trans-unit id="mlang_tabs_tab" xml:space="preserve">
                <source>SB: Comment Admin</source>
            </trans-unit>
        </body>
    </file>
</xliff>
```

15.3. TypoScript

Previously when `pi_base` was used, it was not possible to use TypoScript in the backend of TYPO3 CMS. This has changed with Extbase, modules may (and sometimes must) be configured by TypoScript.

For this purpose, the key `module.tx_[extensionkey]` exists (`[extensionkey]` is the extension key without underscore) so in our case: `module.tx_simpleblog`. Since it often makes sense to re-use plugin configurations as the basis for a new configuration, we can just copy an existing one.

Add the following to the file `typo3conf/ext/simpleblog/Configuration/TypoScript/setup.txt`:

```
module.tx_simpleblog < plugin.tx_simpleblog
```

15.4. Comment Repository

All comments, independently from being Blogs or Posts, are required for our backend module. Therefore we need a comment repository, which we create as file `typo3conf/ext/simpleblog/Classes/Domain/Repository/CommentRespository.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\Domain\Repository;

class CommentRepository extends \TYPO3\CMS\Extbase\Persistence\Repository {

    protected $defaultOrderings = array('commentdate' => \TYPO3\CMS\Extbase\

}
?>
```

As you can see, we have chosen the standard sorting by date, with the newest entry at the top.

15.5. Comment Controller

Next, we create the controller as file `typo3conf/ext/simpleblog/Classes/Controller/CommentController.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\Controller;

class CommentController extends \TYPO3\CMS\Extbase\Mvc\Controller\ActionCont

    /**
     * commentRepository
     *
     * @var \Lobacher\Simpleblog\Domain\Repository\CommentRepository
     * @inject
     */
    protected $commentRepository;


    public function initializeAction(){
        $querySettings = $this->objectManager->get( 'TYPO3\\CMS\\Extbase\\Pe
        $querySettings->setRespectStoragePage(FALSE);
        $querySettings->setIgnoreEnableFields(TRUE);
        $querySettings->setEnableFieldsToBeIgnored(array('disabled'));
        $querySettings->setIncludeDeleted(TRUE);
        $this->commentRepository->setDefaultQuerySettings($querySettings);
    }

    public function listAction() {
        $this->view->assign('commentsLive', $this->commentRepository->findBy
        $this->view->assign('commentsDeleted', $this->commentRepository->fin
    }

    /**
     * @param \Lobacher\Simpleblog\Domain\Model\Comment $comment
     */
    public function deleteAction(\Lobacher\Simpleblog\Domain\Model\Comment $
        $this->commentRepository->remove($comment);
        $this->redirect('list');
    }

    public function testAction() {
       return 'Output of testAction';
    }

}
?>
```

In order to apply the default query settings to the repository, we use the `initializeAction()`, which ensures we retrieve all comments on all pages, independently from their `storagePid` (`setRespectStoragePage(FALSE)`) and including those comments, which have already been deleted (`setRespectEnableFields(FALSE)`). By calling `$this->commentRepository->setDefaultQuerySettings($querySettings)`, these settings are applied to the comment repository and set as defaults. An alternative solution would be to directly configure these settings in the repository.

The action `listAction()` accesses the comment repository twice: first, to retrieve all records, which have the property `deleted` set to `0` (these are the entries not deleted) and second, to retrieve all records with the value set to `1` (these are the deleted entries). Both are passed to the view. An appropriate getter is not required for the property `deleted` because a `findBy[Property]()` call accesses the data storage directly.

Originated at the list view, a comment object can be passed to the `deleteAction()`, where the comment is deleted from the repository and a redirect to the list action initiated.

The remaining `testAction()` simply acts as a dummy method, which can be used to investigate the subject further and extend your knowledge.

15.6. List View

The list view differs from all previously discussed views in regard to the markup but this is not a must necessarily. We only want to integrate the view into the TYPO3 backend smoothly with a focus on the visual appearance.

This begins with the `<f:be.container>` ViewHelper, which features the following API:

```
/**
 * Render start page with \TYPO3\CMS\Backend\Template\DocumentTemplate and p
 *
 * @param string  $pageTitle title tag of the module. Not required by defaul
 * @param boolean $enableJumpToUrl If TRUE, includes "jumpTpUrl" javascript
 * @param boolean $enableClickMenu If TRUE, loads clickmenu.js required by B
 * @param boolean $loadPrototype specifies whether to load prototype library
 * @param boolean $loadScriptaculous specifies whether to load scriptaculous
 * @param string  $scriptaculousModule additionales modules for scriptaculou
 * @param boolean $loadExtJs specifies whether to load ExtJS library. Defaul
 * @param boolean $loadExtJsTheme whether to load ExtJS "grey" theme. Defaul
 * @param string  $extJsAdapter load alternative adapter (ext-base is defaul
 * @param boolean $enableExtJsDebug if TRUE, debug version of ExtJS is loade
 * @param string $addCssFile Custom CSS file to be loaded (deprecated, use $
 * @param string $addJsFile Custom JavaScript file to be loaded (deprecated,
 * @param boolean $loadJQuery whether to load jQuery library. Defaults to FA
 * @param array $includeCssFiles List of custom CSS file to be loaded
 * @param array $includeJsFiles List of custom JavaScript file to be loaded
 * @param array $addJsInlineLabels Custom labels to add to JavaScript inline
 */
```

Even if no further options are defined, the ViewHelper takes care of some default values, for example some JavaScript code is inserted automatically, which is required for a Jump Menu (a menu at the top of the screen, which loads the new content on function selection).

In addition, some specific CSS classes are included, which ensure the proper layout and styles are available for the grey and black areas at the top, e.g. `typo3-fullDoc`, `typo3-docheader` and `typo3-docheader-functions`.

15.6.1. Structure

The structure of a backend markup looks as follows:

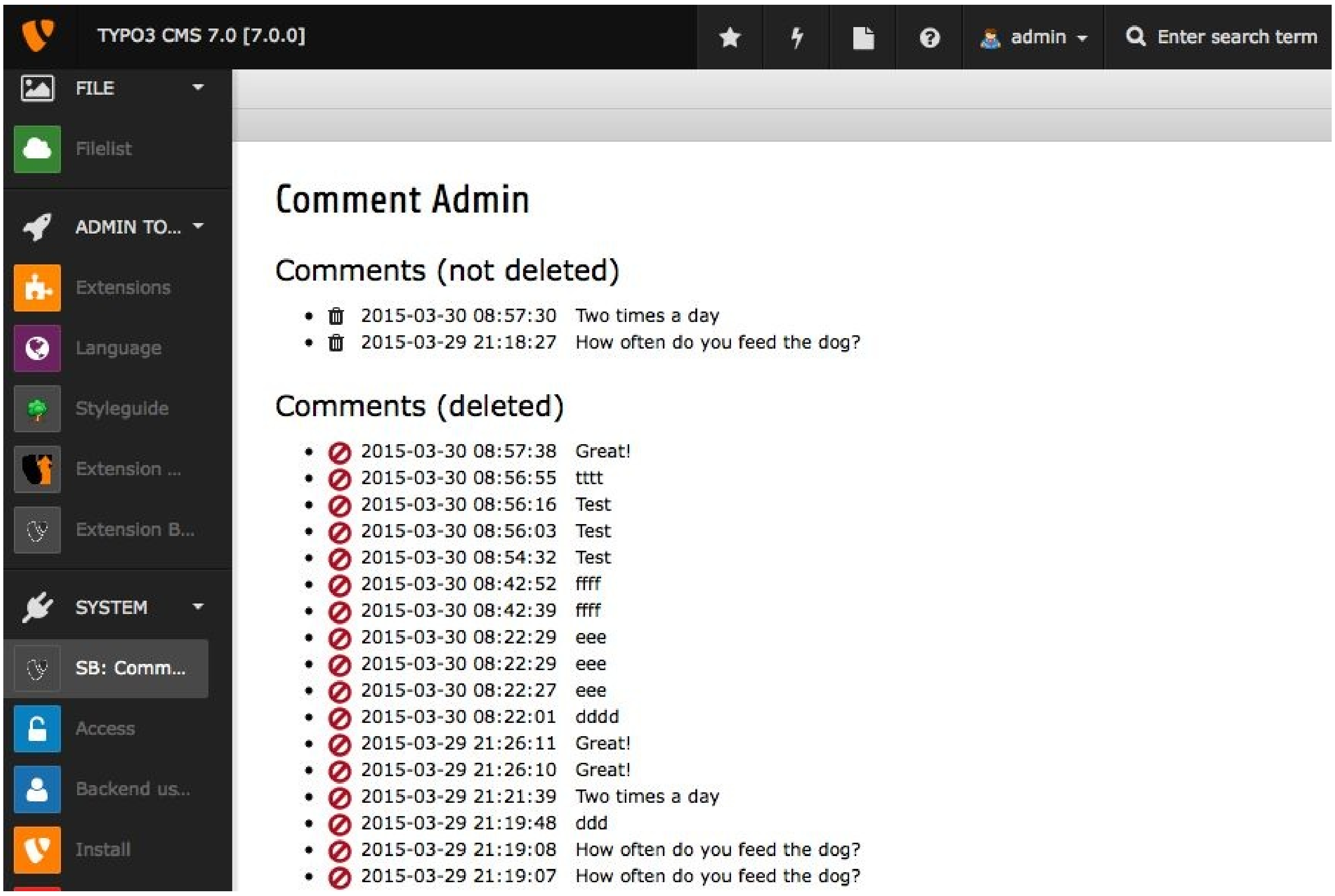
```
<f:be.container>

    <div class="typo3-fullDoc">

        <div id="typo3-docheader">
            <div class="typo3-docheader-functions">
                <div class="left">
                    <f:be.buttons.csh />

                    <f:render section="button-toolbar" optional="true" />
                </div>
                <div class="right">
                </div>
            </div>
            <div class="typo3-docheader-buttons">
                <div class="left">

                </div>
                <div class="right">
                    <f:be.buttons.shortcut />
                </div>
            </div>
        </div>

        <div id="typo3-docbody">
            <div id="typo3-inner-docbody">
                <f:flashMessages renderMode="div" />
                <f:render section="content" />
            </div>
        </div>

    </div>

</f:be.container>
```

15.6.2. Content of the List Template

Based on this basic structure, we can easily build the list action template (file: `typo3conf/ext/simpleblog/Resources/Private/Templates/Comment/List.html`) with the following content:

```
<f:be.container>
    <div class="typo3-fullDoc">
        <div id="typo3-docheader">
            <div class="typo3-docheader-functions">
                <div class="left">
                    <f:be.buttons.csh field="delete" />
                    <div style="margin:-28px 28px;">
                    <f:be.menus.actionMenu>
                        <f:be.menus.actionMenuItem label="Comment Admin" con
                        <f:be.menus.actionMenuItem label="Test" controller="
                    </f:be.menus.actionMenu>
                    </div>
                </div>
                <div class="right">
                </div>
            </div>
            <div class="typo3-docheader-buttons">
                <div class="left">
                    <!-- Left Page -->
                </div>
                <div class="right">
                    <f:be.buttons.shortcut />
                </div>
            </div>
        </div>
        <div id="typo3-docbody">
            <div id="typo3-inner-docbody">
                <h1>Comment Admin</h1>

                <h2>Comments (not deleted)</h2>
                <ul>
                    <f:for each="{commentsLive}" as="comment">
                        <li>
                            <f:link.action controller="Comment" action="dele
                            <f:format.date format="Y-m-d H:i:s">{comment.com
                            {comment.comment}
                        </li>
                    </f:for>
                </ul>

                <h2>Comments (deleted)</h2>
                <ul>
                    <f:for each="{commentsDeleted}" as="comment">
                        <li>
                            <f:be.buttons.icon icon="apps-pagetree-drag-plac
                            <f:format.date format="Y-m-d H:i:s">{comment.com
                            {comment.comment}
                        </li>
                    </f:for>
                </ul>
            </div>
        </div>
    </div>
</f:be.container>
```

Figure 15.1. Comment backend module in action



Comment Admin

Comments (not deleted)

- 2015-03-30 08:57:30 Two times a day
- 2015-03-29 21:18:27 How often do you feed the dog?

Comments (deleted)

- 2015-03-30 08:57:38 Great!
- 2015-03-30 08:56:55 tttt
- 2015-03-30 08:56:16 Test
- 2015-03-30 08:56:03 Test
- 2015-03-30 08:54:32 Test
- 2015-03-30 08:42:52 ffff
- 2015-03-30 08:42:39 ffff
- 2015-03-30 08:22:29 eee
- 2015-03-30 08:22:29 eee
- 2015-03-30 08:22:27 eee
- 2015-03-30 08:22:01 dddd
- 2015-03-29 21:26:11 Great!
- 2015-03-29 21:26:10 Great!
- 2015-03-29 21:21:39 Two times a day
- 2015-03-29 21:19:48 ddd
- 2015-03-29 21:19:08 How often do you feed the dog?
- 2015-03-29 21:19:07 How often do you feed the dog?

This template creates two lists: the first one shows all comments, which have not been deleted yet. An icon with a link enables backend users (administrators) to delete the entry. In addition, the creation date and the text is shown. A click on the icon triggers a request to the `deleteAction` of the controller and deletes the comment without confirmation. The second list shows the already deleted comments.

The next sections explain the used ViewHelpers in more detail.

15.6.3. CSH Buttons

Context Sensitive Help (CSH) allows users to identify and understand certain functionality of the graphical user interface (GUI) quickly. In order to fit our module with this feature, we need to implement the `<f:be.button.csh field="delete" />` ViewHelper. The option `field` reflects a key, which can be found in a language file. This file must be registered in `ext_tables.php`:

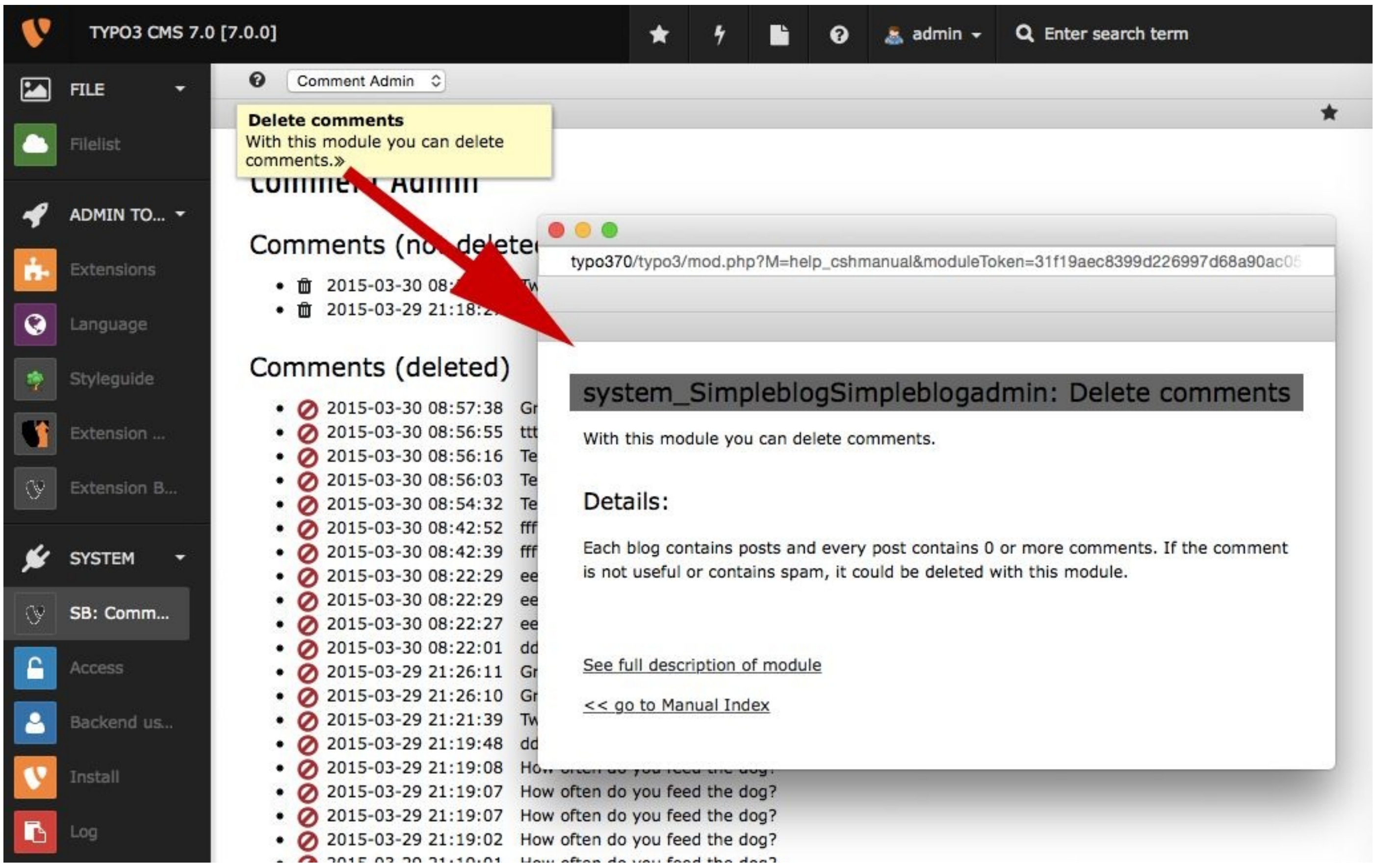
```
\TYPO3\CMS\Core\Utility\ExtensionManagementUtility::addLLrefForTCAdescr(
    '_MOD_system_SimpleblogSimpleblogadmin',
    'EXT:' . $_EXTKEY . '/Resources/Private/Language/locallang_csh.xlf'
);
```

Then, create an XLF file `typo3conf/ext/simpleblog/Resources/Private/Language/locallang_csh.xlf` with the following content:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<xliff version="1.0">
    <file source-language="en" datatype="plaintext" original="messages" date
        <header/>
        <body>
            <trans-unit id="delete.alttitle" xml:space="preserve">
              <source>Delete comments</source>
            </trans-unit>
            <trans-unit id="delete.description" xml:space="preserve">
              <source>With this module you can delete comments.</source>
            </trans-unit>
            <trans-unit id="delete.details" xml:space="preserve">
              <source>Each blog contains posts and every post contains 0 or
            </trans-unit>
        </body>
    </file>
</xliff>
```

Each key (in our case: `delete`) requires three sub-keys: `alttitle` (title at roll over state), `description` (description at roll over state) and `details` (text, which appears when a user clicks the button).

Figure 15.2. Display of context sensitive help



Delete comments
With this module you can delete comments.»

Comment Admin

Comments (not deleted)

- 2015-03-30 08:
- 2015-03-29 21:18:

Comments (deleted)

- 2015-03-30 08:57:38 Gr
- 2015-03-30 08:56:55 ttt
- 2015-03-30 08:56:16 Te
- 2015-03-30 08:56:03 Te
- 2015-03-30 08:54:32 Te
- 2015-03-30 08:42:52 fff
- 2015-03-30 08:42:39 fff
- 2015-03-30 08:22:29 ee
- 2015-03-30 08:22:29 ee
- 2015-03-30 08:22:27 ee
- 2015-03-30 08:22:01 dd
- 2015-03-29 21:26:11 Gr
- 2015-03-29 21:26:10 Gr
- 2015-03-29 21:21:39 Tw
- 2015-03-29 21:19:48 dd
- 2015-03-29 21:19:08 How often do you feed the dog?
- 2015-03-29 21:19:07 How often do you feed the dog?
- 2015-03-29 21:19:07 How often do you feed the dog?
- 2015-03-29 21:19:02 How often do you feed the dog?

typo370/typo3/mod.php?M=help_cshmanual&moduleToken=31f19aec8399d226997d68a90ac0

system_SimpleblogSimpleblogadmin: Delete comments

With this module you can delete comments.

Details:

Each blog contains posts and every post contains 0 or more comments. If the comment is not useful or contains spam, it could be deleted with this module.

See full description of module

<< go to Manual Index

15.6.4. Action Menu

The *Action Menu* is a dropdown box near the CSH icon. This menu allows backend users to switch to specific functions of the module. An outer `<f:be.menus.actionMenu>` ViewHelper wraps an arbitrary number of `<f:be.menus.actionMenuItem>` ViewHelpers. The latter are fitted with a label as well as a controller and an action, which are called when the user selects the appropriate menu item from the list.

```
<f:be.menus.actionMenu>
    <f:be.menus.actionMenuItem label="Comment Admin" controller="Comment" ac
    <f:be.menus.actionMenuItem label="Test" controller="Comment" action="tes
</f:be.menus.actionMenu>
```

15.6.5. Shortcut Button

The well-known shortcut button in TYPO3 can easily be implemented by using the `<f:be.buttons.shortcut />` ViewHelper.

15.6.6. Icon Button

The ViewHelper `<f:be.buttons.icon>` can be used to get the icons of the trash bin (function “delete”) and the stop sign.

```
<f:be.buttons.icon icon="actions-edit-delete" title="Delete comment" />
<f:be.buttons.icon icon="apps-pagetree-drag-place-denied" title="Comment del
```

More than 310 abbreviations are available for the attribute `icon`, which are all listed in file `typo3/systext/core/ext_tables.php`.

An overview of all icons available can be generated by the extension `spiteIconOverview`.[42] Unfortunately, it depends on `extdeveval`, which is currently not available for TYPO3 CMS version 6.2 LTS and above. For this reason, we have created a PDF document that shows all icons and their names.[43]

---

[42] http://typo3.org/extensions/repository/view/spriteiconoverview

[43] http://www.extbase-book.org/resources.html

Chapter 16. The Property Mapper

The Property Mapper predominantly converts simple data types (e.g. arrays, strings or numbers) into objects. This is particularly important in the MVC context, when a HTTP request arrives, which only contains simple types and no objects.

However due to the fact that Extbase has been designed to transport everything via objects, incoming data should be transformed into their object representation. This is the main purpose of the Property Mapper.

The API of the Property Mappers is quite simple: in its core, it consists of a method `convert($source, $targetType)`, which accepts the input data as the first argument and the type, in which the data should be converted, as the second argument.

16.1. Examples

Let’s have a look at an example, which converts a simple type (a string) into a floating point value:

```
// $propertyMapper is an instance of class
// TYPO3\CMS\Extbase\Property\PropertyMapper
$result = $propertyMapper->convert('42.5', 'float');
// $result == (float)42.5
```

A more complex example is the following:

```
class Lobacher\Simpleblog\Domain\Model\Post extends \TYPO3\CMS\Extbase\Domai

    /**
     * @var \string
     */
    protected $title;

    /**
     * @var \DateTime
     */
    protected $postdate;

    /**
     * @var \Lobacher\Simpleblog\Domain\Model\Author
     */
    protected $author;

    ...
}
```

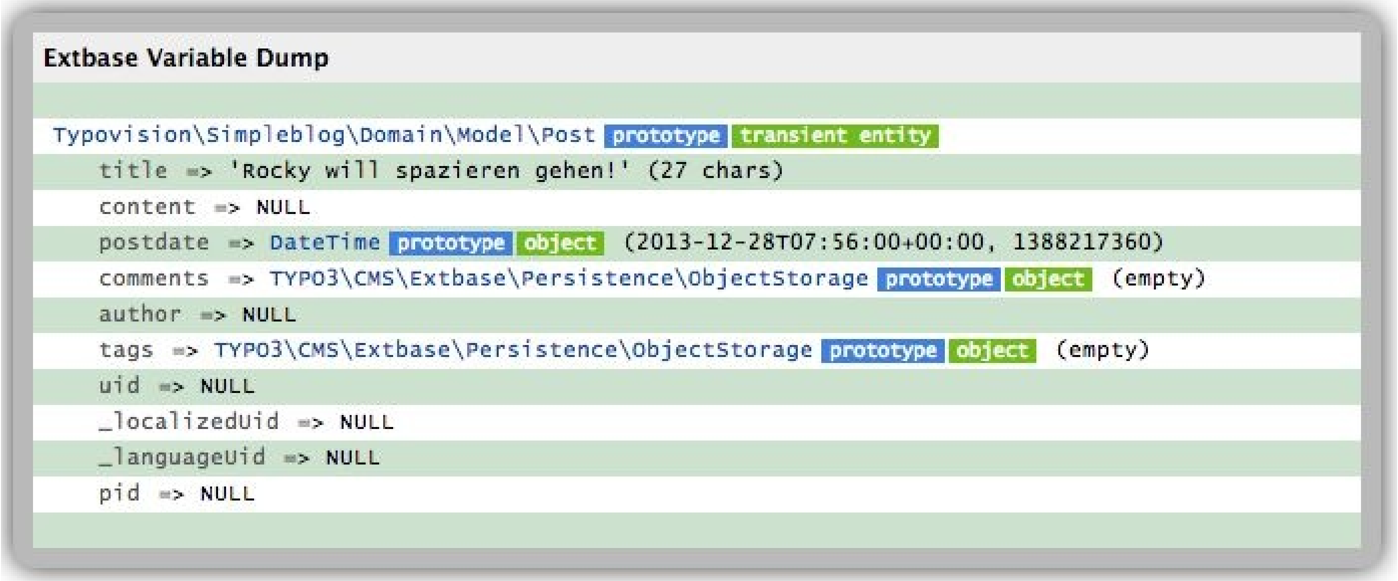
This is an extract of the domain model of a post, which contains a title and a timestamp as properties.

We can convert this data into an object now: add the code below to an action of the Blog controller:

```
$inputArray = array(
    'title' => 'Rocky wants to go for a stroll!',
    'postdate' => '2013-12-28T07:56:00+00:00'
);

$propertyMapper = $this->objectManager->get('TYPO3\\CMS\\Extbase\\Property\\

$post = $propertyMapper->convert(
    $inputArray,
    'Lobacher\Simpleblog\Domain\Model\Post');

\TYPO3\CMS\Extbase\Utility\DebuggerUtility::var_dump($post);
```

The result is an object of class `Lobacher\Simpleblog\Domain\Model\Post`, which has set both properties `title` and `postdate` correctly.

Figure 16.1. Post object



```
Extbase Variable Dump

Typovision\Simpleblog\Domain\Model\Post prototype transient entity
   title => 'Rocky will spazieren gehen!' (27 chars)
   content => NULL
   postdate => DateTime prototype object (2013-12-28T07:56:00+00:00, 1388217360)
   comments => TYPO3\CMS\Extbase\Persistence\ObjectStorage prototype object (empty)
   author => NULL
   tags => TYPO3\CMS\Extbase\Persistence\ObjectStorage prototype object (empty)
   uid => NULL
   _localizedUid => NULL
   _languageUid => NULL
   pid => NULL
```

You may ask, how does the Property Mapper know how to convert objects of type `DateTime` or `Post`? In fact, it does not know this itself but uses specific *TypeConverters*. Currently, about 18 different TypeConverters exist in directory `TYPO3\CMS\Extbase\Property\TypeConverter`. Our example above used three of them:

- First, in order to convert “Rocky wants to go for a stroll!” into a string (required by annotation `@var \string` in the domain model), `StringConverter`: it converts the value into a string. Due to the fact that the output is already a string, it will simply be passed on.
- `DateTimeConverter`, which generates a valid object of type `DateTime` from the value `2013-12-28T07:56:00+00:00`.
- Finally, an object of type `Post` has to be created. The `PersistentObjectConverter` takes care of that by creating a clean, new object of type `Lobacher\Simpleblog\Domain\Model\Post` and setting the properties `$title` and `$postdate` via the setter of the object.

The example shows that the property mapping is a recursive process, where the Property Mapper coordinates the procedure. The `PersistentObjectConverter` has some more features, e.g. it can retrieve objects from the persistence layer, if an object identity has been stated:

Both input values (directly set with its UID or by using the key `__identity`) in the subsequent code result in retrieving the Post with UID 2 from the persistence layer (usually the database) and reconstructing it so that an object `$post` is available at the end.

```
$input = '2';
```

```
// or

$input = array(
  '__identity' => '2'
);

$post = $propertyMapper->convert(
    $input,
    'Lobacher\Simpleblog\Domain\Model\Post'
);
```

If further properties are defined in the array (`besides __identity`), they are modified in the object. However these modifications are not stored at the end of the request but have to be made persistent by calling method `update()` of the repository.

```
$input = array(
  '__identity' => '2',
  'title' => 'This is a new title!',
  'author' => '3'
);
$post = $propertyMapper->convert(
    $input,
    'Lobacher\Simpleblog\Domain\Model\Post'
);
```

The following actions are executed:

- The post with UID `2` is being retrieved from the database.
- The property `$title` is set.
- An object of type `\Lobacher\Simpleblog\Domain\Model\Author` is reconstructed.
- For that, the data of the author with UID `3` is being retrieved from the database.
- This object is being set as the property `$author` of the post object.

16.2. Property Mapper Configuration

It is also possible to configure the conversion process by passing a third parameter `PropertyMappingConfiguration` to the `PropertyMapper::convert()` method. If no `PropertyMappingConfiguration` is specified, the `PropertyMappingConfigurationBuilder` creates a default configuration.

The `PropertyMappingConfigurationBuilder` should be used to generate a new `PropertyMappingConfiguration`:

```
$propertyMappingConfigurationBuilder = $this->objectManager->get( 'TYPO3\\CM

$propertyMappingConfiguration = $propertyMappingConfigurationBuilder->build(

\TYPO3\CMS\Extbase\Utility\DebuggerUtility::var_dump($propertyMappingConfigu

// You can modify $propertyMappingConfiguration now…
// and pass the configuration to convert()
$propertyMapper->convert(
    $source,
    $targetType,
    $propertyMappingConfiguration
);
```

Available options are:

`setMapping($sourcePropertyName, $targetPropertyName)`

: Can be used to rename properties. Assuming, the input array contains a property `lastName` but the associated property of the domain object is `$givenName`, the following method call conducts a renaming:

    ```
    $propertyMappingConfiguration->setMapping('lastName', 'givenName');
    ```

`setTypeConverter($typeConverter)`

: This overwrites the automatic resolution of the Type Converter to apply and forces Extbase to use a specific one.

`setTypeConverterOption($typeConverterClassName, $optionKey, $optionValue)`

: This sets further options for the TypeConverter. For example, the `DateTimeConverter` can be configured to use a specific date format:

    ```
    setTypeConverterOption( 'TYPO3\CMS\Extbase\Property\TypeConverter\DateTi
    ```

`setTypeConverterOptions($typeConverterClassName, array $options)`

: This sets multiple options at the same time.

`allowProperties($propertyName1, $propertyName2, ...)`

This specifies, which properties are allowed on the current level. The usage of the wildcard `*` (all properties) is also valid.

`allowAllProperties()`

This equals `allowProperties('*')` and allows the conversation of all properties.

`allowAllPropertiesExcept($propertyName1, $propertyName2)`

This creates a blacklist of properties, which must not be converted. All properties not included in this list can be converted.

The configuration options are applied on the current level only – without further settings on the top level. In the case they should be extended to sub levels, `forProperty($propertyPath)` can be stated.

```
$propertyMappingConfiguration->setMapping('longTitle', 'title');
$propertyMappingConfiguration
    ->forProperty('postdate')
    ->setTypeConverterOption(
        '\TYPO3\CMS\Extbase\Property\TypeConverter\DateTimeConverter',
        \TYPO3\CMS\Extbase\Property\TypeConverter\DateTimeConverter:: CONFIG
        'Y-m-d'
    );
```

The dot-syntax allows `forProperty()` to support more than one nesting level, e.g. `post.author.name`.

The Property Mapper uses indices as property names for properties, which contain multiple values (e.g. arrays). To achieve a matching of all indices, the wildcard `*` can be used:

```
$propertyMappingConfiguration
    ->forProperty('comments.*')
    ->setTypeConverterOption(
        '\TYPO3\CMS\Extbase\Property\TypeConverter\PersistentObjectConverter
        \TYPO3\CMS\Extbase\Property\TypeConverter\PersistentObjectConverter:
        TRUE
    );
```

16.3. Property Mapper Configuration in MVC Stack

The main field of application of the Property Mapper Configuration is surely the MVC stack, where incoming arguments are converted into objects. If Fluid forms are used, an explicit configuration is not required in most cases, only if a web service or AJAX client is being developed.

Usually, the access happens via the object `TYPO3\CMS\Extbase\Mvc\Controller\Argument`, which is available in the controller as `$this->arguments`. Actions `initializeAction()` or `initialize[ActionName]Action()` are the relevant methods.

The following example reconstructs a comment object from the argument `comment` and uses it as an object in the `update()` action:

```
public function initializeUpdateAction() {

    $commentConfiguration = $this->arguments['comment']->getPropertyMappingC

    $commentConfiguration->allowAllProperties();
    $commentConfiguration
        ->setTypeConverterOption(
            '\TYPO3\CMS\Extbase\Property\TypeConverter\PersistentObjectConve
            \TYPO3\CMS\Extbase\Property\TypeConverter\PersistentObjectConver
            TRUE
    );
}

/**
 * @param \Lobacher\Simpleblog\Domain\Model\Comment $comment
 */
public function updateAction(\Lobacher\Simpleblog\Domain\Model\Comment $comm
        // object $comment can be used now
        ...
}
```

IDE conform syntax

Some IDE have issues with the following syntax:

```
$commentConfiguration = $this->arguments['comment']->getPropertyMapping
```

This is because they can not resolve the method after an array access (type hinting). In this case, the following alternative syntax can be used:

```
    $commentConfiguration = $this->arguments->getArgument('comment')->getPr
```

16.4. Security Aspects

Let’s assume, a user should be able to create a new account via a REST API and also set a role (out of a list of available roles).

```
array(
  'username' => 'newusername',
  'role' => '2'
);
```

An attacker could manipulate the array as follows:

```
array(
  'username' => 'newusername',
  'role' => array(
    'name' => 'superuser',
    'admin' => 1
  )
);
```

Due to the fact that the Property Mapper works recursively, a new role object would be created and its admin flag set.

For this reason, the recursive behaviour must be configured at two spots:

- Allowed properties must be specified by `allowProperties()`, `allowAllProperties()` or `allowAllPropertiesExcept()`, and:
- The `PersistentObjectConverter` must be configured with options `CONFIGURATION_MODIFICATION_ALLOWED` and `CONFIGURATION_CREATION_ALLOWED`. Only this configuration enables the creation or modification of objects. By default, the `PersistentObjectConverter` can retrieve objects from the database but can not create new or modifies existing ones.

16.5. API Reference

The Property Mapper goes through the following steps to convert simple data types into objects:

- It determines the TypeConverter in order to convert the source into the target.
- It determines the child properties of the source (if exists) by calling method `getSourceChildPropertiesToBeConverted()`.
- Then, for each child property:
  - it determines the data type of each child property by calling method `getTypeOfChildProperty()`,
  - recursive call of the Property Mapper in order to reconstruct all child objects.

It calls the TypeConverter again, by passing all reconstructed objects so far to method `convertFrom()`.

- The result is the final object tree.

16.5.1. Automatic Resolution of TypeConverter

All TypeConverters, which implement the interface `\TYPO3\CMS\Extbase\Property\TypeConverterInterface`, are found automatically during the resolving process. Each TypeConverter features four API methods, which impact this process:

`getSupportedSourceTypes()`
: Returns an array of source types, which the TypeConverter can handle.

`getSupportedTargetType()`
: Returns the target type this TypeConverter converts to. This can be a simple type (e. g. `float`) or a FQCN (e.g. `'TYPO3\\CMS\\Extbase\\Domain\\Model\\File'`).

`getPriority()`
: If two TypeConverters have the same source and target type, the priority specifies, which one has precedence. TypeConverters with a high priority are chosen before low priority and all default TypeConverter have a priority of 100 or less.

`canConvertFrom($source, $targetType)`
: With this method the TypeConverter can do some additional runtime checks to see whether it can handle the given source data and the given target type.

Chapter 17. Best Practices

This chapter explains some typical challenges and their practical solutions. In no particular order of importance, we’ll take a closer look at Flash Messages, AJAX, image uploads, RSS feeds and a range of other possible problem areas.

17.1. Flash Messages

Extbase implements a number of actions as an HTTP request “travels” through and many things may happen along this journey. It’s useful to be informed on what exactly takes place during this journey. The issue, however, is that there is no specific point where Extbase stops in order to output an update, so the only solution is to gather the details and display them at the journey’s end (the last action).

This is the purpose of so-called *Flash Messages*. They are stored in a container (respectively in a queue since TYPO3 CMS 6.1) until they are retrieved. To display these messages a ViewHelper can be used which also empties the container/queue and allows new Flash Messages to be added again.

System-wide Flash Messages do not require any configuration but in order to get messages on a plugin-only basis, the following TypoScript must be set:

```
config.tx_extbase.legacy.enableLegacyFlashMessageHandling = 0
```

At this point, a new message can be added, e.g. when a new Blog has been entered and stored in file `typo3conf/ext/simpleblog/Classes/Controller/BlogController.php`:

```
...
    public function addAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog)
        $this->addFlashMessage(
                'Blog successfully created!',
                'Status',
                \TYPO3\CMS\Core\Messaging\AbstractMessage::OK,TRUE
            )
        );
        $this->blogRepository->add($blog);
        $this->redirect('list');
    }
...
```

The API behind this method call reads as follows:

```
addFlashMessage(
    $messageBody,
    $messageTitle = '',
    $severity = \TYPO3\CMS\Core\Messaging\AbstractMessage::OK,
    $storeInSession = TRUE
);
```

Possible levels of severity are:

- `NOTICE`
- `INFO`

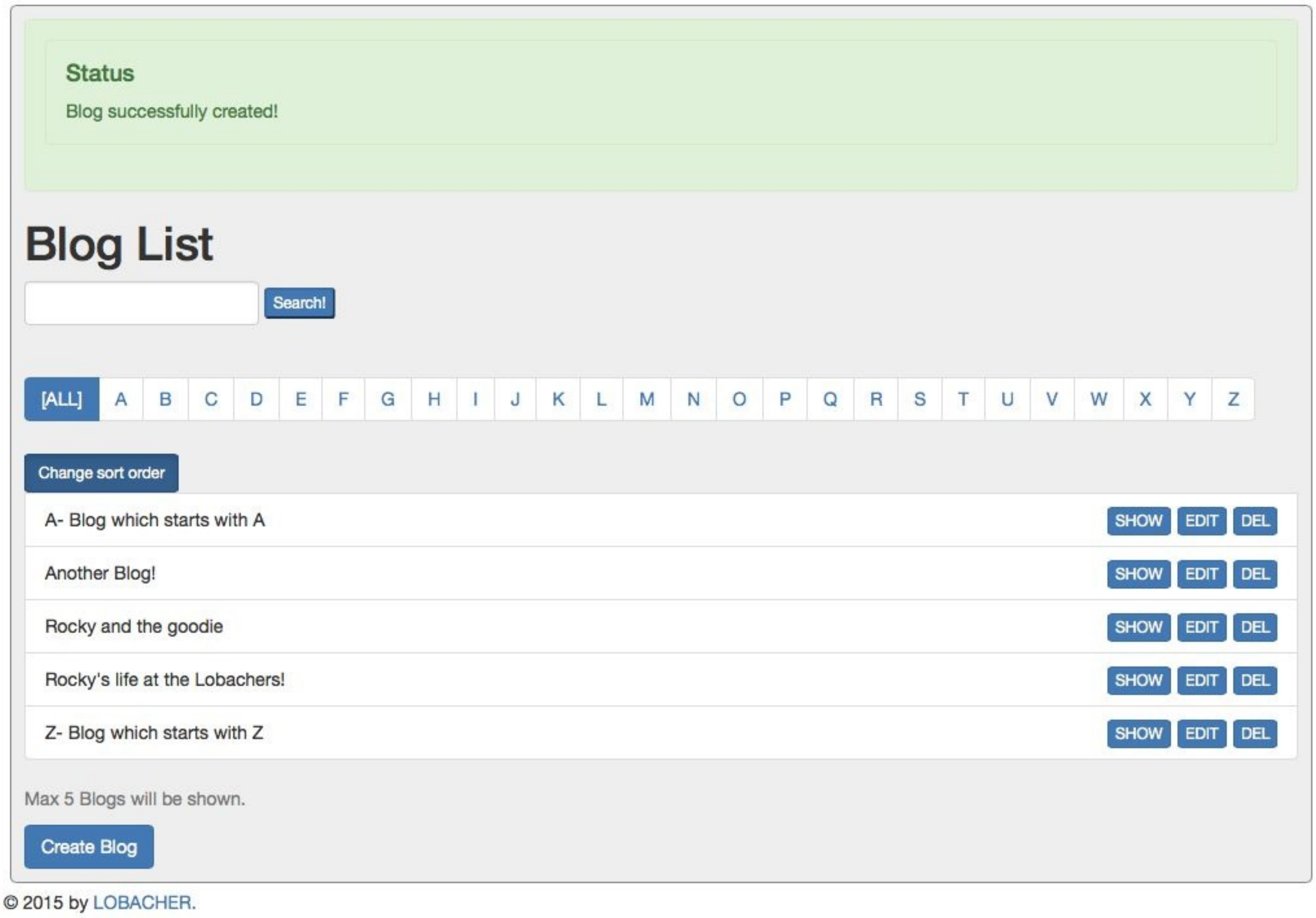
- `OK`
- `WARNING`
- `ERROR`

They are called from the class `\TYPO3\CMS\Core\Messaging\AbstractMessage` for example: `\TYPO3\CMS\Core\Messaging\AbstractMessage::OK`.

In order to display Flash Messages you can use your own ViewHelper. We insert it in the list action of the Blog in file `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/List.html`:

```
...
<f:section name="content">
...
<f:flashMessages renderMode="div" class="alert alert-success" />
...
```

Figure 17.1. Output of a Flash Message


Status

Blog successfully created!

Blog List

A- Blog which starts with A	SHOW EDIT DEL
Another Blog!	SHOW EDIT DEL
Rocky and the goodie	SHOW EDIT DEL
Rocky's life at the Lobachers!	SHOW EDIT DEL
Z- Blog which starts with Z	SHOW EDIT DEL

Max 5 Blogs will be shown.

© 2015 by LOBACHER.

With just a few lines of jQuery code you could also set the message to disappear again after a few seconds.

17.2. Load Plugin Via TypoScript

Extbase itself loads the plugin via TypoScript, making it very easy for us to do the same – for example, to include the plugin on every page without the need to add it as a content element.

An example:

```
10 = USER
10 {
    userFunc = TYPO3\CMS\Extbase\Core\Bootstrap->run
    extensionName = Simpleblog
    pluginName = Bloglisting
    vendorName = Lobacher
    controller = Blog
    action = rss
    switchableControllerActions {
        Blog {
            1 = rss
        }
    }
    settings =< plugin.tx_simpleblog.settings
    persistence =< plugin.tx_simpleblog.persistence
    view =< plugin.tx_simpleblog.view
}
```

`extensionName`
: Contains the name of the extension (in UpperCamelCase).

`pluginName`
: Specifies the plugin name.

`vendorName`
: Contains the vendor name.

`controller`
: If you want to jump into a specific controller, name it here.

`action`
: If you want to execute a specific action, name it here.

`switchableControllerActions`
: Allows access to be restricted to specific controller/action combinations for this implementation. The next subkey represents the controller name and subsequently the action as a number (starting at `1`) and the name of the action.

`settings`
: Sets the settings which can be copied and/or referenced.

`persistence`
: Sets the persistence settings which can be copied and/or referenced.

`view`
: Sets the view settings which can be copied and/or referenced.

17.3. RSS-Feed

In a perfect world it would not be necessary to check if there are any new Posts in our Blog; we would use an RSS Feed to inform us automatically instead. This is what we will implement in the step below.

First, position a link in the show template of your Blog:

```
<f:link.action action="rss" format="xml" pageType="100" arguments="{blog:blo
```

The `<f:link>` ViewHelper refers to the action `rss`, which we have to implement later. Additionally, we force the format to be `xml` rather than the default `html` and request page type `100` explicitly, which means a custom TypoScript rendering is required. Finally, we add the Blog as an argument to the request, so we can show its posts in the RSS feed.

In the next step we edit file `typo3conf/ext/simpleblog/ext_localconf.php` and enable the new action `rss`:

```
\TYPO3\CMS\Extbase\Utility\ExtensionUtility::configurePlugin(
    'Lobacher.' . $_EXTKEY,
    'Bloglisting',
    array(
        'Blog' => '...,delete,rss',
        ...
    ),
    // non-cacheable actions
    array(
        'Blog' => '...,delete,rss',
        ...
    )
);
```

Now we extend the Blog controller `typo3conf/ext/simpleblog/Classes/Controller/BlogController.php` by adding the appropriate method:

```
    /**
     * RSS Feed for the posts of one blog
     *
     * @param \Lobacher\Simpleblog\Domain\Model\Blog $blog
     */
    public function rssAction(\Lobacher\Simpleblog\Domain\Model\Blog $blog)
        $this->view->assign('blog', $blog);
    }
```

We also have to create the file `typo3conf/ext/simpleblog/Resources/Private/Templates/Blog/Rss.xml` to show the

RSS feed:

```
<feed xmlns="http://www.w3.org/2005/Atom">
<author>
    <name>Author of the Weblog</name>
</author>
<title>{blog.title}</title>
<id>{f:uri.action(action:'show', controller:'Blog', arguments:'{blog:blog}',
<updated><f:format.date format='Y-m-d\TH:i:sP'>{blog.crdate}</f:format.date>

<f:for each="{blog.posts}" as="post">
<entry>
    <title>{post.title}</title>
    <link href="{f:uri.action(action:'show', controller:'Post', arguments:'{
    <id>{f:uri.action(action:'show', controller:'Post', arguments:'{post:pos
    <updated><f:format.date format='Y-m-d\TH:i:sP'>{post.postdate}</f:format
    <summary>{post.content -> f:format.crop(maxCharacters:'30')}</summary>
    <content>{post.content}</content>
</entry>
</f:for>

</feed>
```

In order to read the `crdate`, the TCA and the Blog model need to be expanded. Let’s start with the TCA and edit the file `typo3conf/ext/simpleblog/Configuration/TCA/Blog.php`:

```
...
        'crdate' => Array (
            'exclude' => 0,
            'label' => 'Creation date',
            'config' => Array (
                'type' => 'none',
                'format' => 'date',
                'eval' => 'date',
            )
        ),
...
```

The property `crdate` and the appropriate getter as well as the setter are added to the Blog model:

```
...
    /**
     * crdate
     * @var DateTime
     */
    protected $crdate;

    /**
     * @param DateTime $crdate
     * @return void
     */
    public function setCrdate(DateTime $crdate) {
        $this->crdate = $crdate;
    }

    /**
     * @return DateTime
     */
    public function getCrdate() {
        return $this->crdate;
    }
...
```

Finally, we implement the page type in TypoScript:

```
rss = PAGE
rss {
    typeNum = 100
    10 = USER
    10 {
        userFunc = TYPO3\CMS\Extbase\Core\Bootstrap->run
        extensionName = Simpleblog
        pluginName = Bloglisting
        vendorName = Lobacher
        controller = Blog
        action = rss
        switchableControllerActions {
            Blog {
                1 = rss
            }
        }
        settings =< plugin.tx_simpleblog.settings
        persistence =< plugin.tx_simpleblog.persistence
        view =< plugin.tx_simpleblog.view
    }
    config {
        disableAllHeaderCode = 1
        additionalHeaders = Content-type:application/xml
        xhtml_cleaning = 0
        admPanel = 0
    }
}
```

17.4. UriBuilder

Extbase comes with its own UriBuilder that enables developers to generate URLs based on specific factors such as the current action, controller, required UID, etc. You can easily build links in your controller or ViewHelper analogue to the ViewHelper `<f:link.action>`.

```
...
$uriBuilder = $this->controllerContext->getUriBuilder();
$uri = $uriBuilder->uriFor('show',array('blog'=>$blog),'Blog');
...
```

Method `uriFor()` has the following parameters:

```
public function uriFor($actionName = NULL, $controllerArguments = array(), $
```

In order to generate a “standard link” further methods can be called to provide granular control:

```
$uriBuilder = $this->controllerContext->getUriBuilder();
$uri = $uriBuilder->reset()
            ->setTargetPageUid($pageUid)
            ->setTargetPageType($pageType)
            ->setNoCache($noCache)
            ->setUseCacheHash(!$noCacheHash)
            ->setSection($section)
            ->setFormat($format)
            ->setLinkAccessRestrictedPages($linkAccessRestrictedPages)
            ->setArguments($additionalParams)
            ->setAbsoluteUriScheme($absoluteUriScheme)
            ->setCreateAbsoluteUri($absolute)
            ->setAddQueryString($addQueryString)
            ->setArgumentsToBeExcludedFromQueryString( $argumentsToBeExclude
            ->setAddQueryStringMethod($addQueryStringMethod)
            ->build();
```

Every method (such as `setFormat()`) is also available as a reading method (`getFormat()`).

The example above also shows a reset of all the parameters used in order to build the URL completely from scratch.

The UriBuilder can also be used to determine the current URL:

```
$this->uriBuilder->getRequest()->getRequestUri());
```

17.5. File Upload

As part of the domain model we also allowed for a property `image` in the Blog `object`. We have ignored this until now however, because Extbase does not yet offer a native function to deal with such uploads.

We will change this now and bring this feature to life. This requires some steps:

17.5.1. Adding an Upload Field

To begin, we extend the `<f:form>` ViewHelper in the file `typo3conf/ext/simpleblog/Resources/Private/Partials/Blog/Form.html` and add a (conditional) `image` ViewHelper as well as an upload field:

```
<f:form enctype="multipart/form-data" action="{action}" object="{blog}" name
...
    <div class="form-group">
        <label>Blog-Bild</label>
        <f:if condition="{blog.image}"><br /><f:image src="uploads/tx_simple
        <f:form.upload  property="image" class="form-control" />
    </div>

    <f:form.submit value="{submitmessage}" class="btn btn-primary" />
</f:form>
```

Due to the fact that the image ViewHelper only stores the file name in `{blog.image}`, we can simply use a `<f:if>` ViewHelper to check if an image exists.

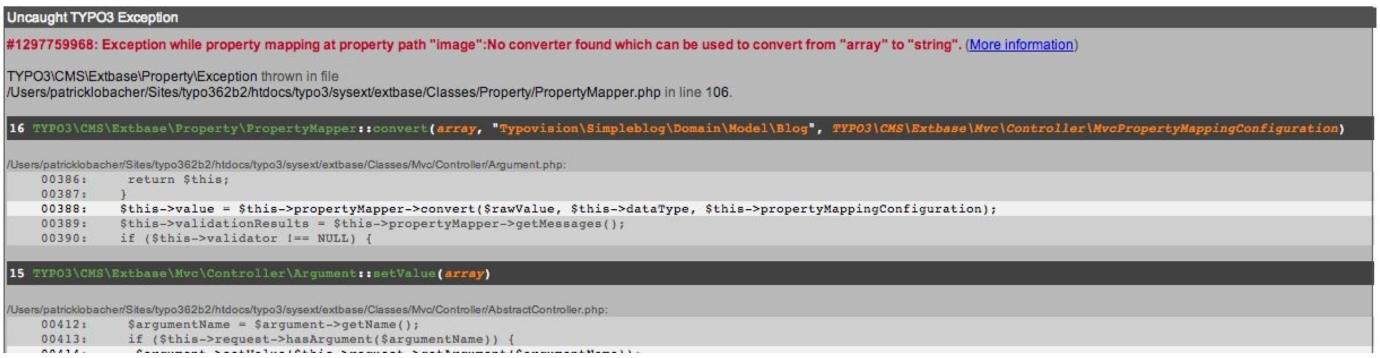
17.5.2. Blog Controller Adjustments

As soon as someone selects an image and submits the form the following error message appears:

```
#1297759968: Exception while property mapping at property path "image":No co
```

The reason for this is that an upload is sent as an array internally (file name, temporary name, etc.) but the property expects a `string` instead.

Figure 17.2. Error message at Property Mapper



```
Uncaught TYPO3 Exception
#1297759968: Exception while property mapping at property path "image":No converter found which can be used to convert from "array" to "string". (More information)

TYPO3\CMS\Extbase\Property\Exception thrown in file
/Users/patricklobacher/Sites/typo362b2/htdocs/typo3/sysext/extbase/Classes/Property/PropertyMapper.php in line 106.

16 TYPO3\CMS\Extbase\Property\PropertyMapper::convert(array, "Typovision\Simpleblog\Domain\Model\Blog", TYPO3\CMS\Extbase\Mvc\Controller\MvcPropertyMappingConfiguration)

/Users/patricklobacher/Sites/typo362b2/htdocs/typo3/sysext/extbase/Classes/Mvc/Controller/Argument.php:
         return $this;
        }
        $this->value = $this->propertyMapper->convert($rawValue, $this->dataType, $this->propertyMappingConfiguration);
        $this->validationResults = $this->propertyMapper->getMessages();
        if ($this->validator !== NULL) {

15 TYPO3\CMS\Extbase\Mvc\Controller\Argument::setValue(array)

/Users/patricklobacher/Sites/typo362b2/htdocs/typo3/sysext/extbase/Classes/Mvc/Controller/AbstractController.php:
         $argumentName = $argument->getName();
         if ($this->request->hasArgument($argumentName)) {
```

An example array could look like the following:

```
array(5) {
  ["name"]=>
  string(14) "tv_twitter.png"
  ["type"]=>
  string(9) "image/png"
  ["tmp_name"]=>
  string(26) "/private/var/tmp/phpmmDkhX"
  ["error"]=>
  int(0)
  ["size"]=>
  int(3033)
}
```

We have learnt already that this scenario requires a TypeConverter – for example in the `initializeAction()` of the Blog controller

`typo3conf/ext/simpleblog/Classes/Controller/BlogController.php`:

```
    public function initializeAction(){
        if ($this->arguments->hasArgument( 'blog' )) {
            $this->arguments->getArgument( 'blog' )->getPropertyMappingConfi
        }
    }
```

Assuming an argument blog `exists`, attribute `image` is converted before the Blog object is passed on to the action.

17.5.3. Blog Model Adjustments

At the time of the upload the `image` is represented as an array, however a string is required (simply the file name). In theory we could make this conversion take place in the controller but this is business logic which belongs to the model.

Instead, we adjust the setter of property image in the Blog model. To do so, edit the file `typo3conf/ext/simpleblog/Classes/Domain/Model/Blog.php`:

```
/**
 * Sets the image
 *
 * @param \array $image
 * @return void
 */
public function setImage(array $image) {
    if (!empty($image['name'])) {
        // image name
        $imageName = $image['name'];
        // temporary name (incl. path) in upload directory
        $imageTempName = $image['tmp_name'];
        $basicFileUtility = \TYPO3\CMS\Core\Utility
    \GeneralUtility:: makeInstance('TYPO3\\CMS\\Core\\Utility\\File\\Bas
        // determining a unique namens (incl. path) in
        // uploads/tx_simpleblog/ and copy file
        $imageNameNew = $basicFileUtility->getUniqueName( $imageName, \T
    \TYPO3\CMS\Core\Utility\GeneralUtility:: upload_copy_move($imageTemp
        // set name without path
        $this->image = basename($imageNameNew);
    }
}
```

17.5.4. TCA Adjustments

Last but not least, we have to update the TCA by editing the file `typo3conf/ext/simpleblog/Configuration/TCA/Blog.php`:

```
...
        'image' => array(
            'exclude' => 0,
            'label' => 'LLL:EXT:simpleblog/Resources/Private/Language/locall
            'config' => array(
                'type' => 'group',
                'internal_type' => 'file',
                'uploadfolder' => 'uploads/tx_simpleblog',
                'show_thumbs' => 1,
                'size' => 1,
                'allowed' => $GLOBALS['TYPO3_CONF_VARS']['GFX']['imagefile_e
                'disallowed' => '',
            ),
        ),
...
```

This declares that exactly one file can be assigned to image (`size`) and that a thumbnail (preview image) appears in the backend (`show_thumbs`).

Upload via FAL (File Abstraction Layer)

Intentionally, the example above does not use the FAL (*File Abstraction Layer*) which is part of the TYPO3 CMS core since version 6.0. The reason is because the write process is not yet fully implemented. Currently, reading files is classified as stable with FAL and is well documented in the TYPO3 Wiki for example.[44]

17.6. StdWrap in Settings

Our current TypoScript settings do not support any stdWrap functionalities. It would be great if something like the following worked:

```
plugin.tx_simpleblog {
    settings {
        uid.dataWrap = {page:uid}
    }
}
```

However this would not be parsed and processed. To allow such dynamic data the settings array needs to be converted into a dot-syntax in the controller and the stdWrap function applied to it:

```
// load TypoScript service class
$typoScriptService = $this->objectManager->get('TYPO3\\CMS\\Extbase\\Service

// convert settings into TypoScript array
$settingsAsTypoScriptArray = $typoScriptService->convertPlainArrayToTypoScri

// apply stdWrap
$this->settings['uid'] = $this->configurationManager->getContentObject()->st
```

17.7. Signal Slot Dispatcher

The Signal Slot concept is a design pattern in the software engineering which implements an event-driven program flow – specifically event-driven communication between objects. Originally introduced by the framework Qt, this concept is used by a number of other software libraries today.

The pattern is an application of the design pattern *Observer* and can be seen as an alternative to direct callback functions.

Generally speaking, an object can send (emit) a signal at any time, e.g. when a specific event occurs, for example “added to the repository”. On the other hand, slots are functions, which are associated with one or multiple signals and are executed automatically, if one of those signals is emitted.

The main area of application of Signal and Slots is in the expansion of extensions by other extensions – similar to Hooks or the XCLASS concept of the traditional extension development with `pi_base()`.

Signals are emitted by a so-called Dispatcher which has the following syntax:

```
$this->signalSlotDispatcher->dispatch(
    $signalClassName,
    $signalName,
    array $signalArguments = array()
);
```

A practical example could be:

```
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'afterInsertObject',
    array('object' => $object)
);
```

PHP’s predefined “magical” constant `__CLASS__` defines the current class name followed by the Signal name and all arguments which are given to the signal to take with them.

Then the Slot function is connected to the Signal and with that the function is called automatically when the Signal is emitted:

```
$this->signalSlotDispatcher->connect(
    $signalClassName,
    $signalName,
    $slotClassNameOrObject,
    $slotMethodName = '',
    $passSignalInformation = TRUE);
```

A practical example could be:

```
$this->signalSlotDispatcher->connect(
    'TYPO3\\CMS\\Extbase\\Persistence\\Generic\\Backend',
    'afterUpdateObject',
    'Lobacher\\Simpleblog\\Service\\SignalService',
    'handleUpdateEvent',
    TRUE
);
```

Arguments are the class name – which emits the Signal – followed by the Signal name. Then follow the name of the Slot class and the appropriate method, which will be executed as soon as the Signal is emitted. The last argument defines any additional information that should be passed, e.g. `$signalClassName . '::' . $signalName` (for example: `TYPO3\CMS\Extbase\Persistence\Generic\Backend::afterUpdateObject`).

17.7.1. Built-in Signals

Extbase comes with numerous built-in Signals which can be used straight away.

In the file `TYPO3\CMS\Extbase\Persistence\Generic\Backend`:

```
// emits a signal before object data is fetched
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'beforeGettingObjectData',
    array($query)
);

// emits a signal after object data is fetched
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'afterGettingObjectData',
    array($query, $result)
);

// emits a signal after an object is added to the storage
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'afterInsertObject',
    array('object' => $object)
);

// emits a signal after an object is updated in storage
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'afterUpdateObject',
    array('object' => $object)
);

// emits a signal after an object is removed from storage
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'afterRemoveObject',
    array('object' => $object)
);
```

In file `TYPO3\CMS\Extbase\Mvc\Dispatcher`:

```
// emits a signal after a request is dispatched
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'afterRequestDispatch',
    array('request' => $request, 'response' => $response)
);
```

In file `TYPO3\CMS\Extbase\Mvc\Controller\ActionController`:

```
// emits a signal before the current action is called
$this->signalSlotDispatcher->dispatch(
    __CLASS__,
    'beforeCallActionMethod',
    array(
        'controllerName' => get_class($this),
        'actionMethodName' => $this->actionMethodName,
        'preparedArguments' => $preparedArguments
    )
);
```

17.7.2. Example Usage of Built-in Signals

As an example of how to use built-in Signals we will write a log file entry as soon as a new comment has been added. For that we use the Signal `afterInsertObject`, which we have to connect to a Slot in the file `ext_localconf.php`:

```
$signalSlotDispatcher = \TYPO3\CMS\Core\Utility\GeneralUtility::makeInstance
    'TYPO3\\CMS\\Extbase\\SignalSlot\\Dispatcher'
);

$signalSlotDispatcher->connect(
    'TYPO3\\CMS\\Extbase\\Persistence\\Generic\\Backend',
    'afterInsertObject',
    'Lobacher\\Simpleblog\\Service\\SignalService',
    'handleInsertEvent'
);
```

As soon as the Signal `afterInsertObject` is emitted the method `handleUpdateEvent` of class `Lobacher\Simpleblog\Service\SignalService` is called.

We need to create the file `typo3conf/ext/simpleblog/Classes/Service/SignalService.php` with the following content:

```
<?php
namespace Lobacher\Simpleblog\Service;

class SignalService implements \TYPO3\CMS\Core\SingletonInterface {

    /**
     * @param \TYPO3\CMS\Extbase\DomainObject\DomainObjectInterface $object
     */
    public function handleInsertEvent(\TYPO3\CMS\Extbase\DomainObject\Domain
        if ($object instanceof \Lobacher\Simpleblog\Domain\Model\Comment) {
            $content = 'Comment: '. $object->getComment();
            $content .= ' / ' . $object->getCommentdate()->format('Y-m-d H:i
            $content .= " / " . $signalInformation . chr(10);
            $this->writeLogFile($content);
        }
    }

    /**
     * @param $content string
     */
    public function writeLogFile($content){
        $logfile = "logfile.txt";
        $handle = fopen($logfile, "a+");
        fwrite ($handle, $content);
        fclose ($handle);
    }

}
```

```
?>
```

The current object is transferred to the method `handleUpdateEvent()` as `$object`. The first step is to check if the object is of the type `comment`. If this is the case, a string `$content` is created, which shows the following content written to the file `logfile.txt`:

```
Comment: This is a comment / 2013-12-27 13:18:25 / TYPO3\CMS\Extbase\Persist
```

This also shows from where the Signal originated: `TYPO3\CMS\Extbase\Persistence\Generic\Backend::afterInsertObject` (Class::SignalName).

Parameter modifications

It is not only possible to read data in a Slot, but we can also update them. The comment object could also be modified by executing `$object->setComment('New comment')` for example.

17.7.3. Create Your Own Signals

With the knowledge gained so far it becomes very easy to implement your own Signals. This is especially important if your extensions should be expandable by other extensions. Whenever it makes sense to extend functionality, implement a Signal and document this clearly. This enables other developers who use your extension to add features by writing a Slot, without modifying the code of your extension.

Let’s add a Signal to the Post controller of the action `ajax` to allow others to “expand” comments at this level as required.

To achieve this, edit the file `typo3conf/ext/simpleblog/Classes/Controller/PostController.php`:

```
class PostController extends \TYPO3\CMS\Extbase\Mvc\Controller\ActionControl
    ...
    /**
     * SignalSlotDispatcher
     *
     * @var \TYPO3\CMS\Extbase\SignalSlot\Dispatcher
     * @inject
     */
    protected $signalSlotDispatcher;
    ...
    public function ajaxAction(
        \Lobacher\Simpleblog\Domain\Model\Post $post,
        \Lobacher\Simpleblog\Domain\Model\Comment $comment = NULL) {

        $comment->setCommentdate(new \DateTime());
        $post->addComment($comment);

        // signal for comments
        $this->signalSlotDispatcher->dispatch(
            __CLASS__,
            'beforeCommentCreation',
            array($comment,$post)
        );

        $this->postRepository->update($post);
        ...
    }
    ...
}
```

First of all, we make sure that the Signal Slot Dispatcher is available by using Dependency Injection. Then we implement its method `dispatch()` in our `ajaxAction()` in order to emit a Signal with the name `beforeCommentCreation` and the parameters `$comment` and `$post`.

At this point we are able to receive and process the Signal in any extension we choose. For the sake of simplicity we will do this in our own `Simpleblog` extension.

Edit file `ext_localconf.php`:

```
$signalSlotDispatcher->connect(
    'Lobacher\\Simpleblog\\Controller\\PostController',
    'beforeCommentCreation',
    'Lobacher\\Simpleblog\\Service\\SignalService',
    'handleCommentInsertion'
);
```

Finally, we need a method `handleCommentInsertion()` in the file `typo3conf/ext/simpleblog/Classes/Service/SignalService.php`, which we have defined as the Signal name in the method call `connect()` before.

```
    /**
     * @param \TYPO3\CMS\Extbase\DomainObject\DomainObjectInterface $comment
     * @param \TYPO3\CMS\Extbase\DomainObject\DomainObjectInterface $post
     * @param $signalInformation string
     */
    public function handleCommentInsertion(
            \TYPO3\CMS\Extbase\DomainObject\DomainObjectInterface $comment,
            \TYPO3\CMS\Extbase\DomainObject\DomainObjectInterface $post,
            $signalInformation){
        $content = 'Comment: '. $comment->getComment();
        $content .= ' (Post: ' . $post->getTitle() . ')';
        $content .= " / " . $signalInformation . chr(10);
        $this->writeLogFile($content);
    }
```

This results in a log entry such as:

```
Comment: My owner always works on his book rather than going for a stroll! (

Comment: My owner always works on his book rather than going for a stroll! /
```

The first entry originates from the Signal `beforeCommentCreation` (and the appropriate Slot `handleCommentInsertion()`), the second from the Signal `afterInsertObject` (and the appropriate Slot `handleInsertEvent()`).

17.8. File Abstraction Layer (FAL)

Since TYPO3 CMS version 6.0 the *File Abstraction Layer* (FAL) is used to manage all kinds of media files, such as images or videos. While write access to the FAL is not yet fully supported by Extbase, reading data is not a problem and a convenient way to deal with multimedia files.

A new column in the database table is required in order to equip the model with a FAL image field:

```
images int(11) unsigned DEFAULT '0',
```

Then, we have to reference the field in the appropriate TCA of the model:

```
...
    'images' => array(
        'exclude' => 0,
        'label' => 'images',
        'config' => \TYPO3\CMS\Core\Utility\ExtensionManagementUtility::getF
            'images',
            array(
                'appearance' => array(
                    'headerThumbnail' => array(
                        'width' => '100',
                        'height' => '100',
                    ),
                'createNewRelationLinkTitle' => 'LLL:EXT:your_extension/Reso
            ),
            // custom configuration for displaying fields in the overlay/ref
            // to use the imageoverlayPalette instead of the basicoverlayPal
            'foreign_types' => array(
                '0' => array(
                    'showitem' => '
                        --palette--;LLL:EXT:lang/locallang_tca.xlf:sys_file_
                        --palette--;;filePalette'
                ),
                \TYPO3\CMS\Core\Resource\File::FILETYPE_TEXT => array(
                    'showitem' => '
                        --palette--;LLL:EXT:lang/locallang_tca.xlf:sys_file_
                        --palette--;;filePalette'
                ),
                \TYPO3\CMS\Core\Resource\File::FILETYPE_IMAGE => array(
                    'showitem' => '
                        --palette--;LLL:EXT:lang/locallang_tca.xlf:sys_file_
                        --palette--;;filePalette'
                ),
                \TYPO3\CMS\Core\Resource\File::FILETYPE_AUDIO => array(
                    'showitem' => '
                        --palette--;LLL:EXT:lang/locallang_tca.xlf:sys_file_
                        --palette--;;filePalette'
                ),
                \TYPO3\CMS\Core\Resource\File::FILETYPE_VIDEO => array(
                    'showitem' => '
                        --palette--;LLL:EXT:lang/locallang_tca.xlf:sys_file_
                        --palette--;;filePalette'
                ),
                \TYPO3\CMS\Core\Resource\File::FILETYPE_APPLICATION => array
                    'showitem' => '
                        --palette--;LLL:EXT:lang/locallang_tca.xlf:sys_file_
                        --palette--;;filePalette'
                )
            ),
        ),
        $GLOBALS['TYPO3_CONF_VARS']['GFX']['imagefile_ext']
    )
),
...
```

The model itself must be extended by the property and the getter and setter methods:

```
/**
 * FAL Image Connector
 * @var \TYPO3\CMS\Extbase\Persistence\ObjectStorage<\TYPO3\CMS\Extbase\Doma
 * @lazy
 */
protected $images;

/**
 * Constructor
 * @return AbstractObject
 */
public function __construct() {
   ...
   $this->images = new \TYPO3\CMS\Extbase\Persistence\ObjectStorage();
   ...
}

/**
 * Images Setter
 * @param \TYPO3\CMS\Extbase\Persistence\ObjectStorage $images
 * @return void
 */
public function setImages($images) {
   $this->images = $images;
}

/**
 * Images Getter
 * @return \TYPO3\CMS\Extbase\Persistence\ObjectStorage
 */
public function getImages() {
   return $this->images;
}
```

After these few preparations images can be selected and assigned via FAL in the backend of TYPO3 and displayed in a Fluid template:

```
<f:for each="{images}" as="image" >
   <a href="{f:uri.image(src:image.uid,treatIdAsReference:1)}" class="lightb
      <f:image src="{image.uid}" alt="" width="100" height="50" treatIdAsRef
   </a >
</f:for >
```

Usually, a title “Create new relation” is shown in the backend where the user has the option to add images via FAL. You can alter this title by the following code (for example in the file `ext_tables.php`):

```
\TYPO3\CMS\Core\Utility\ExtensionManagementUtility::getFileFieldTCAConfig('i
   array(
      'appearance' => array(
         'headerThumbnail' => array(
            'width' => '100',
            'height' => '100',
         ),
         'createNewRelationLinkTitle' => 'LLL:EXT:your_extension/Resources/P
       )
   ),
   'jpg,jpeg,png');
```

Providing the following conditions are met:

- `image` contains the FAL object.
- `imageoriginalResource` contains the `sys_file_reference` data record.
- `imageoriginalResource.originalFile` contains the `sys_file` data record.

You can access the FAL data by the keywords listed below:

```
File name:    {image.originalResource.originalFile.name}
Title:        {image.originalResource.originalFile.title}
Description:  {image.originalResource.originalFile.description}
Alt text:     {image.originalResource.originalFile.alternative}
UID:          {image.originalResource.originalFile.uid}
Path:         {image.originalResource.publicUrl}

Reference attributes:
Title:        {image.originalResource.title}
Description:  {image.originalResource.name}

Display as an image:
<f:image src="{image.originalResource.originalFile.uid}" alt="" />
<f:image src="{image.uid}" alt="" treatIdAsReference="TRUE" />
```

17.9. Category API

Extbase features a so-called category API since TYPO3 CMS version 6.0. To enable data records to be able to be categorised the following API code can be used:

```
\TYPO3\CMS\Core\Utility\ExtensionManagementUtility::makeCategorizable(
    $extensionKey,
    $tableName,

    // optional: in case the field would need a different name as "categorie
    // The field is mandatory for TCEmain to work internally.
    $fieldName = 'categories',

    // optional: add TCA options which controls how the field is displayed.
    $options = array()
);
```

You can choose between using the Extension Builder or the manual method to add categories to your own model.

For the Extension Builder the following steps are required:

- create a model, which should be categorised;
- add a `1:n` relation with the name `categories`;
- in field *Relation to external class*, enter: `\TYPO3\CMS\Extbase\Domain\Model\Category`
- extend file `ext_tables.php` by the API call `\TYPO3\CMS\Core\Utility\ExtensionManagementUtility::makeCategorizable()` (see above) to make the field categorisable;
- modify the model TCA and delete the configuration, which has been added by the Extension Builder automatically for the field `categories`; and,
- access categories via getter and setter directly.

Alternatively, this is the manual procedure:

```
/**
 * Categories
 *
 * @var \TYPO3\CMS\Extbase\Persistence\ObjectStorage<\TYPO3\CMS\Extbase\Doma
 */
protected $categories;

/**
 * Adds a Category
 *
 * @param \TYPO3\CMS\Extbase\Domain\Model\Category $category
 * @return void
 */
public function addCategory(\TYPO3\CMS\Extbase\Domain\Model\Category $catego
    $this->categories->attach($category);
}

/**
 * Removes a Category
 *
 * @param \TYPO3\CMS\Extbase\Domain\Model\Category $categoryToRemove The Cat
 * @return void
 */
public function removeCategory(\TYPO3\CMS\Extbase\Domain\Model\Category $cat
    $this->categories->detach($categoryToRemove);
}

/**
 * Returns the categories
 *
 * @return \TYPO3\CMS\Extbase\Persistence\ObjectStorage<\TYPO3\CMS\Extbase\D
 */
public function getCategories() {
    return $this->categories;
}

/**
 * Sets the categories
 *
 * @param \TYPO3\CMS\Extbase\Persistence\ObjectStorage<\TYPO3\CMS\Extbase\Do
 * @return void
 */
public function setCategories(\TYPO3\CMS\Extbase\Persistence\ObjectStorage $
    $this->categories = $categories;
}
```

Further details can be found in the Wiki of the TYPO3 project.[45]

17.10. Extbase Models

Extbase already contains a number of models, which can be found in folder `typo3conf/sysext/extbase/Classes/Domain/Model/`.

- Backend user (`BackendUser.php`)
- Backend usergroup (`BackendUserGroup.php`)
- Frontend user (`FrontendUser.php`)
- Frontend usergroup (`FrontendUserGroup.php`)
- Categories (`Category.php`)
- Files (`File.php`)
- Filemounts (`FileMount.php`)
- File references (`FileReference.php`)
- Folders (`Folder.php`)
- Folder-based file collections (`FolderBasedFileCollection.php`)
- Static file collections (`StaticFileCollection.php`)

17.11. Scheduler Tasks

Extbase can also be used for Scheduler Tasks. The main only thing to you should be aware of here is the fact that you are not working within TYPO3's internal automatismsbuilt-in automatic features and you might therefore, you possibly have to do some things manually. The following example shows a Scheduler Task, that deletes files automatically after a certain period of time.

First of all, register the Scheduler Task in file `ext_localconf.php` as follows:

```
$GLOBALS['TYPO3_CONF_VARS']['SC_OPTIONS']['scheduler']['tasks']['Lobacher
\\Simpleblog\\Command\\FileCommandController'] = array(
   'extension' => $_EXTKEY,
   'title' => 'Filecenter Delete Files',
   'description' => 'Deletes Files in Filecenter after the defined period of
   'additionalFields' => ''
);
```

After that, create a new file `FileCommandController.php` in the directory `typo3conf/ext/simpleblog/Classes/Command/`:

```
<?php
namespace Lobacher\Simpleblog\Command;

/**
 * Class FileCommandController
 *
 * Deletes files after a certain period of time
 */
class FileCommandController extends \TYPO3\CMS\Scheduler\Task\AbstractTask {

    public function execute() {

        // Fetch ObjectManager
        $objectManager = \TYPO3\CMS\Core\Utility\GeneralUtility::
makeInstance('TYPO3\\CMS\\Extbase\\Object\\ObjectManager');

        // instantiate Repository
        $repository = $objectManager->get('Lobacher\\Simpleblog\\Domain\\Rep

        // fetch configuration
        $configurationManager = $objectManager->get('TYPO3\\CMS\\Extbase\\Co
        $settings = $configurationManager->getConfiguration( \TYPO3\CMS\
Extbase\Configuration\ConfigurationManagerInterface::CONFIGURATION_TYPE_
FULL_TYPOSCRIPT);

        $storagePid = $settings['plugin.']['tx_simpleblog.']
['persistence.']['storagePid'];
```

```
        // set query settings (PID)
        $querySettings = $objectManager->get('TYPO3\\CMS\\Extbase\\
Persistence\\Generic\\Typo3QuerySettings');
        $querySettings->setStoragePageIds(array($storagePid));
        $repository->setDefaultQuerySettings($querySettings);

        // access Repository
        $files = $repository->findAll();

        // ... delete files…

        return count($files);
    }

}
```

The only things left are to set the appropriate repository and some TypoScript.

17.12. JSON View

The feature “JsonView” has been ported from TYPO3 Flow to TYPO3 CMS in version 6.2 LTS. When using AJAX or specifically when developing a web service, the controller expects a particular data format that is easy to process.

The JSON format is quite popular because it is lightweight and easy to parse. You could also implement JSON by a Fluid template, but a “real” view is much more practical. In order to use the JSON view in the controller, it must be enabled with the variable `$defaultViewObjectName`.

```
class FooController extends ActionController {
    /**
     * @var string
     */
     protected $defaultViewObjectName = 'TYPO3\CMS\Extbase\Mvc\View\JsonView
     # …
}
```

The assignment happens as before:

```
/**
 * @param \Acme\Demo\Model\Product $product
 * @return void
 */
public function showAction(Product $product) {
    $this->view->assign('value', $product);
}
```

Variable `value` can be rendered now. To render other variables, they have to be configured first:

```
$this->view->setVariablesToRender(array('articles'));
$this->view->assign('articles', $this->articleRepository->findAll());
```

The output looks like the following then:

```
{"name":"Arabica","weight":1000,"price":23.95}
```

You can configure the array in a very granular way:

```
$this->view->assign('value', $product);
$this->view->setConfiguration(ARRAY);
```

The following syntax is a valid configuration for the `ARRAY`:

```
array(
    'value' => array(
        // rendering of property "name" of the object value
        '_only' => array('name')
    ),
    'anothervalue' => array(
        // rendering of all properties, except "password"
        '_exclude' => array('password')
        // additionally, sub-object "address" should be included as a
        // nested JSON object
        '_descend' => array(
            'address' => array(
                // here you can use _only, _exclude und _descend again
            )
        )
    ),
    'arrayvalue' => array(
        // descend into all sub objects
        '_descendAll' => array(
            // here you can use _only, _exclude und _descend again
        )
    ),
    'valueWithObjectIdentifier' => array(
        // the object identifier is not included in the output by
        // default, but can be added if required
        '_exposeObjectIdentifier' => TRUE,
        // the object identifier should not be rendered as "__identity",
        // but as "guid"
        '_exposedObjectIdentifierKey' => 'guid'
    )
)
```

Possible options are:

`_only` (array)
: include the listed proporties only

`_exclude` (array)
: include all properties, except properties listed

`_descend` (associative array)
: include specified sub-objects

`_descendAll` (array)
: include all sub objects (as a numeric array)

`_exposeObjectIdentifier` (boolean)
: include the object identifier as `__identifier`

`_exposeObjectIdentifierKey` (string)
: JSON field name

---

[44] http://wiki.typo3.org/FAL#Usage_in_Extbase_.28in_progress.29

[45] http://wiki.typo3.org/TYPO3_6.0#Category

Part I. Appendix

Appendix A. Reference

A.1. ext_emconf.php

The file `ext_emconf.php` provides all extension configurations required for the TYPO3 Extension Repository (TER) and the Extension Manager. An array `$EM_CONF[$_EXTKEY]` contains key/value pairs:

```
$EM_CONF[$_EXTKEY] = array (
   'key' => 'value',
   ...
);
```

The following keys exist:

`title` (string, mandatory)
: Extension name (in English).

`description` (string, mandatory)
: Short description of functionality (in English).

`category` (string)
: `be` (backend), `module` (backend module), `fe` (frontend), `plugin` (frontend plugin), `misc` (miscellaneous), `services` (TYPO3 services), `templates` (website template), `example` (example), `doc` (documentation).

`version` (`main.sub.dev`)
: Version of the extension. Automatically managed by the Extension Manager. The Format is `[int].[int].[int]`.

`constraints` (array)

```
    array(
      'depends' =>
        array(
          'php' => '5.4.0-5.5.99',
          'typo3' => '6.2.0-7.999.999',
        ),
      'conflicts' =>
        array(
        ),
      'suggests' =>
        array(
        ),
    ),
```

`dependencies` lists extensions, which have to be installed; `conflicts` lists extensions which conflict with the extension; `suggests` lists extensions which use useful and should be installed, too. The version string `6.2.0-7.999.999` includes all version from 6.2.x to 7.x.x.

`state` (string)

one of the following strings: `alpha`, `beta`, `stable`, `experimental`, `test`, `obsolete` or `excludeFromUpdates`.

`uploadFolder` (boolean)
: If set to `1`, a folder named `uploads/tx_[extensionkey]` should be present (`extensionkey` without underscore).

`createDirs` (string)
: Comma-separated directories to be created upon extension installation.

`clearCacheOnLoad` (boolean)
: If set to `1`, the Extension Manager will request TYPO3 to clear the cache when this extension is loaded.

`author` (string)
: Extension author's name.

`author_email` (string)
: Extension author's email address.

`author_company` (string)
: Extension author's company name.

Other options are likely outdated/deprecated and should not be used. The official TYPO3 core API reference[46] lists all valid configuration options and explains their purpose and usage.

A.2. Flexform Field Types

A.2.1. Text Field

```
<label>Text Field</label>
<config>
  <type>input</type>
  <size>20</size>
  <max>30</max>
  <eval>trim</eval>
</config>
```

A.2.2. Date Field

```
<label>Date Field</label>
<config>
  <type>input</type>
  <size>8</size>
  <max>8</max>
  <eval>date</eval>
  <checkbox>1</checkbox>
</config>
```

A.2.3. Checkbox

```
<label>Checkbox</label>
<config>
  <type>check</type>
</config>
```

A.2.4. Textarea

```
<label>Textarea</label>
<config>
  <type>text</type>
  <cols>24</cols>
  <rows>3</rows>
</config>
```

A.2.5. Textarea with RTE

```
<label>Textarea</label>
<config>
  <type>text</type>
  <cols>24</cols>
  <rows>3</rows>
  <defaultExtras>richtext[*]:rte_transform[mode=ts_css]</defaultExtras>
</config>
```

A.2.6. Radio Buttons

```
<label>Radio Buttons</label>
<config>
  <type>radio</type>
  <items type="array">
    <numIndex index="0" type="array">
      <numIndex index="0">label1</numIndex>
      <numIndex index="1">value1</numIndex>
    </numIndex>
    <numIndex index="1" type="array">
      <numIndex index="0">label2</numIndex>
      <numIndex index="1">value2</numIndex>
    </numIndex>
    <numIndex index="3" type="array">
      <numIndex index="0">label3</numIndex>
      <numIndex index="1">value3</numIndex>
    </numIndex>
  </items>
</config>
```

A.2.7. Selectbox

```
<label>Selectbox</label>
<config>
  <type>select</type>
  <items type="array">
    <numIndex index="0" type="array">
      <numIndex index="0">label1</numIndex>
      <numIndex index="1">value1</numIndex>
    </numIndex>
    <numIndex index="1" type="array">
      <numIndex index="0">label2</numIndex>
      <numIndex index="1">value2</numIndex>
    </numIndex>
    <numIndex index="3" type="array">
      <numIndex index="0">label3</numIndex>
      <numIndex index="1">value3</numIndex>
    </numIndex>
  </items>
</config>
```

A.2.8. Selectbox (Multi-Select)

```
<label>Selectbox Multi-Select</label>
<config>
  <type>select</type>
  <items type="array">
    <numIndex index="0" type="array">
      <numIndex index="0">label1</numIndex>
      <numIndex index="1">value1</numIndex>
    </numIndex>
    <numIndex index="1" type="array">
      <numIndex index="0">label2</numIndex>
      <numIndex index="1">value2</numIndex>
    </numIndex>
    <numIndex index="3" type="array">
      <numIndex index="0">label3</numIndex>
      <numIndex index="1">value3</numIndex>
    </numIndex>
  </items>
  <maxitems>3</maxitems>
  <size>3</size>
</config>
```

A.2.9. Page Browser

```
<label>Page Browser</label>
<config>
  <type>group</type>
  <internal_type>db</internal_type>
  <allowed>pages</allowed>
  <size>1</size>
  <maxitems>1</maxitems>
  <minitems>0</minitems>
  <show_thumbs>1</show_thumbs>
</config>
```

---

[46] http://docs.typo3.org/typo3cms/CoreApiReference/ExtensionArchitecture/DeclarationFile/Index.html

Index