

▼ Getting Started

Abstract This tutorial gets you started with FLOW3. The most important concepts such as the MVC framework, object management, persistence and templating are explained on the basis of a sample application.

Author Robert Lemke robert@typo3.org

Copyright 2009 Robert Lemke, licensed under the Creative Commons Attribution-Share Alike 3.0 License

▼ Chapter 1: Introduction

▼ 1 What's FLOW3?

FLOW3 is a PHP-based application framework. It is especially well-suited for enterprise-grade applications and explicitly supports Domain-Driven Design, a powerful software design philosophy. Convention over configuration, Test-Driven Development, Continuous Integration and an easy-to-read source code are other important principles we follow for the development of FLOW3. Needless to say, FLOW3 provides you with a full-stack MVC framework for building state-of-the-art web applications. More exciting though are the first class Dependency Injection support and the Aspect-Oriented Programming capabilities which can be used without a single line of configuration.

▼ 2 What's in this tutorial?

This tutorial explains all the steps to get you started with your very own first FLOW3 project.

Please bring your own computer, a reasonable knowledge of PHP and HTML and at least some initial experience with object-oriented programming. In return you'll surely get some new insights into modern programming paradigms and how to produce clean code in no time.

Note Please note that this tutorial refers to an alpha version of FLOW3 1.0 which means that changes to the API are still possible. However, we're quite confident that the concepts described in this tutorial will remain the same in FLOW3 1.0.0 final.

If you're stuck at some point or stumble over some weirdnesses during the tutorial, please let us know! We appreciate any feedback in our mailing lists, as a ticket in our issue tracker or via private email

Footnote I'll read any feedback you send me to robert@typo3.org but I don't always manage to answer quickly.

Tip This tutorial goes best with a Caffè Latte or, if it's afternoon or late night already, with a few shots of Espresso ...

▼ Chapter 2: Requirements

FLOW3 is being developed and tested on multiple platforms and pretty easy to set up. Nevertheless we recommend that you go through the following list before installing FLOW3, because a server with exotic php.ini settings or wrong file permissions can easily spoil your day.

▼ 1 Server Environment

Without much surprise you'll need a web server for running your FLOW3-based web application. We recommend Apache (though IIS might work too – we just haven't tested it yet). Please make sure that the [mod_rewrite module](#) is enabled.

FLOW3's persistence mechanism and content repository require a [PDO compatible database](#). By default we use SQLite which is bundled with the standard PHP distribution and doesn't require any further setup from your side. In a production context you'll rather want to use MySQL, PostgreSQL or the like.

▼ 2 PHP

FLOW3 was one of the first PHP projects taking advantage of namespaces and other features introduced in PHP version 5.3.0. Because PHP 5.3.0 is not widely installed on web servers, we created [setup guides for the most popular platforms](#) for your convenience.

The default settings and extensions of the PHP distribution should work fine with FLOW3 but it doesn't hurt checking if the PHP modules `mbstring` and `pdo_sqlite` are enabled, especially if you compiled PHP yourself. You should (not only because of FLOW3) turn off magic quotes in your `php.ini` (`magic_quotes_gpc = off`).

The development context and especially the testrunner need more than the default amount of memory. At least during development you should raise the memory limit to about 250 MB in your `php.ini` file.

▼ Chapter 3: Installation

▼ 1 FLOW3 Download

The most recent FLOW3 release can be obtained from <http://flow3.typo3.org/download/> as a `.tgz`, `.zip` or `.bz2` archive. For the purpose of this tutorial we recommend that you download the special *Getting Started* distribution which not only contains FLOW3 but also resources you might need while trying out the steps mentioned in this tutorial. If you want to start your own application from scratch you'll prefer the FLOW3 base distribution.

Once you downloaded the tutorial distribution just unpack the archive in a directory of your choice, e.g. like this for the gzipped tar archive:

```
mkdir -p /var/apache2/htdocs/tutorial+  
tar xzf FLOW3-GettingStarted-1.0.0-alpha3.tgz /var/apache2/htdocs/tutorial/
```

On Windows you create a directory (e.g. `c:\xampp\htdocs\tutorial`), move the `.zip` file into the new directory and unzip it with the Windows Explorer.

The FLOW3 distributions can also be checked out from our Subversion repository. The following Unix command would download the Getting Started distribution:

```
svn export https://svn.typo3.org/FLOW3/Distributions/GettingStarted/tags/1.0.0-alpha3 \  
/var/apache2/htdocs/tutorial/
```

Note Throughout this tutorial we assume that you installed the FLOW3 Getting Started distribution in `/var/apache2/htdocs/tutorial` and that `/var/apache2/htdocs` is the document root of your web server. On a Windows machine you might use `c:\xampp\htdocs` instead.

▼ 2 Directory Structure

Let's take a look at the directory structure of a FLOW3 application:

▼ *Directory structure of a FLOW3 application*

Directory	Description
Configuration/	Application specific configuration, grouped by contexts
Data/	Persistent and temporary data, including caches, logs and database
Packages/	Contains sub directories which in turn contain package directories
Packages/Framework/	Packages which are part of the official FLOW3 distribution
Packages/Application/	Application specific packages
Web/	Public web root

A FLOW3 application usually consists of the above directories. As you see, most of them contain data which is specific to your application, therefore upgrading the FLOW3 distribution is a matter of replacing `Packages/Framework/` by a new release.

FLOW3 is a package based system which means that all code, documentation and other resources are bundled in packages. Each package has its own directory with a defined sub structure. Your own PHP code and resources will usually end up in a package residing below `Packages/Application/`. You're free to create additional directories or symbolic links in `Packages/`, a common one would be called `Shared/` which points to packages shared by multiple applications.

Tip On Unix-like machines it is a good idea to use symbolic links pointing to your own packages which are used in multiple projects. With this strategy you assure that only one master copy of the package exists and avoid the hassle of diverging copies which contain a few changes here and some fixes there.

▼ 3 File Permissions

Most of the directories and files must be readable and writable for the user you're running FLOW3 with. This user will usually be the same one running your web server (`httpd`, `www` or `_www` on most Unix based systems). However it can and usually will happen that FLOW3 is launched from the command line by a different user. Therefore it is important that both, the web server user and the command line user are members of a common group and the file permissions are set accordingly.

We recommend setting ownership of directories and files to the web server's group. All users who also need to launch FLOW3 must also be added this group. Setting the correct permissions is easily done with a little script delivered with the FLOW3 distribution:

```
setfilepermissions.sh commandlineuser webuser webgroup
```

In practice you'll use the script like this:

```
myhost:~ johndoe$ cd /var/apache2/htdocs/tutorial
myhost:tutorial johndoe$ sudo ./Packages/Framework/FLOW3/Scripts/setfilepermissions.sh
johndoe _www _www
```

Now that the file permissions are set, all users who plan using FLOW3 from the command line need to join the web server's group. On a Linux machine this can be done by typing:

```
sudo usermod -a -G _www johndoe
```

On a Mac you can add a user to the web group with the following command:

```
sudo dscl . -append /Groups/_www GroupMembership johndoe
```

You will have to exit your shell / terminal window and open it again for the new group membership to take effect.

Note In this example the web user was `_www` and the web group is called `_www` as well (that's the case on a Mac using [MacPorts](#)). On your system the user or group might be `www-data`, `httpd` or the like - make sure to find out and specify the correct user and group for your environment.

▼ 4 Web Server Configuration

As you have seen previously, FLOW3 uses a directory called `web` as the public web root. We highly recommend that you create a virtual host which points to this directory and thereby assure that all other directories are not accessible from the web. For testing purposes on your local machine it is okay (but not very convenient) to do without a virtual host, but don't try that on a public server!

▼ 4.1 Setting Up a Virtual Host

Assuming that you chose Apache2 as your web server, simply create a new virtual host by adding the following directions to your Apache configuration (`conf/extra/httpd-vhosts.conf` on many systems):

```
<VirtualHost *:80>
  DocumentRoot /var/apache2/htdocs/tutorial/Web/
  ServerName tutorial.local
</VirtualHost>
```

This virtual host will later be accessible via the URL `http://tutorial.local`.

Because FLOW3 provides an `.htaccess` file with `mod_rewrite` rules in it, you need to make sure that the directory grants the necessary rights:

```
<Directory /var/apache2/htdocs/tutorial/>
  AllowOverride FileInfo
</Directory>
```

▼ 4.2 Configure a Context

As you'll learn soon, FLOW3 can be launched in different *contexts*, the most popular being `Production`, `Development`, `Testing` and `Staging`. Although there are various ways to choose the current context, the most convenient is to setup a dedicated virtual host defining an environment variable. Just add the following virtual host to your Apache configuration:

```
<VirtualHost *:80>
  DocumentRoot /var/apache2/htdocs/tutorial/Web/
  ServerName dev.tutorial.local
  SetEnv FLOW3_CONTEXT Development
</VirtualHost>
```

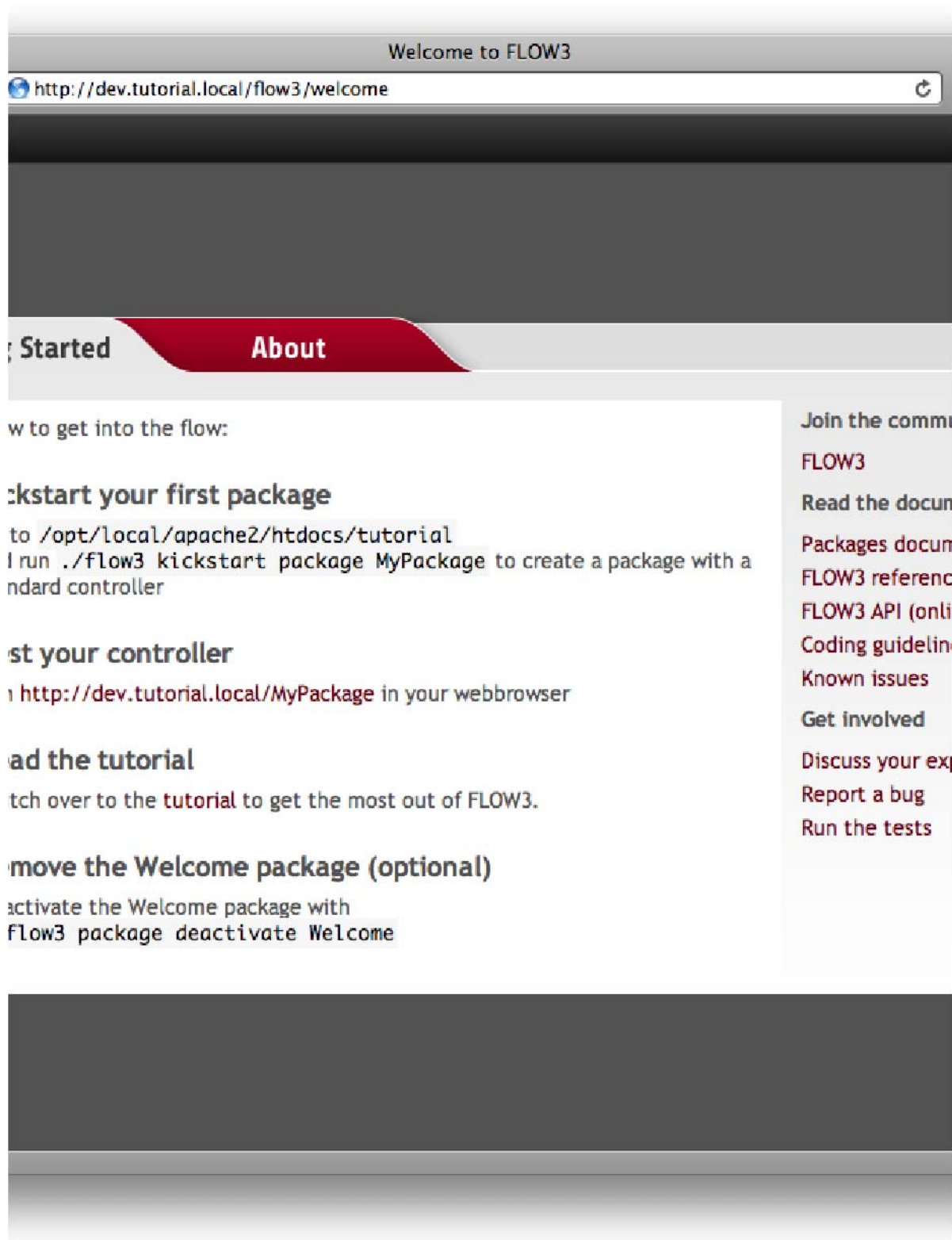
You'll be able to access the same application running in `Development` context by accessing the URL

`http://dev.tutorial.local`. What's left is telling your operating system that the invented domain names can be found on your local machine. Add the following line to your `/etc/hosts` file (C:\windows\system32\drivers\etc\hosts on Windows):

```
127.0.0.1 tutorial.local dev.tutorial.local
```

▼ 4.3 Welcome to FLOW3

Restart Apache and test your new configuration by accessing `http://dev.tutorial.local` in a web browser. You should be greeted by FLOW3's welcome screen:



▼ Chapter 4: Configuration

▼ 1 Contexts

Once you start developing an application you'll want to launch it in different contexts: in a production context the configuration must be optimized for speed and security while in a development context debugging capabilities and convenience are more important. FLOW3 supports the notion of contexts which allow for bundling configuration for different purposes. Each FLOW3 request acts in exactly one context. However, it is possible to use the same installation on the same server in distinct contexts by accessing it through a different host name, port or passing special arguments.

Why do I want contexts?

Imagine your application is running on a live server and your customer reports a bug. No matter how hard you try, you can't reproduce the issue on your local development server. Now contexts allow you to enter the live application on the production server in a development context without anyone noticing - both contexts run in parallel. This effectively allows you to debug an application in its realistic environment (although you still should do the actual development on a dedicated machine ...).

An additional use for context is the simplified staging of your application. You'll want almost the same configuration on your production and your development server - but not exactly the same. The live environment will surely access a different database or might require other authentication methods. What you do in this case is sharing most of the configuration and define the difference in dedicated contexts which might be called `Production_Local / Production_Live, Development_Jamy` or any other name you find meaningful.

Bottomline is: You create a new context by just using its name.

By default FLOW3 provides configuration for the Production, Development, Testing, and Staging context (more contexts may be defined by just adding configuration for it accordingly). In the standard distribution a reasonable configuration is defined for each context:

- In the *Production context* all caches are enabled, logging is reduced to a minimum and only generic, friendly error messages are displayed to the user (more detailed descriptions end up in the log).
- The *Staging context* is targeted at testers and customers who are supposed to give feedback. The application basically behaves like in production context with the difference that error messages and logging are more verbose.
- In *Development context* caches are active but a smart monitoring service flushes caches automatically if PHP code or configuration has been altered. Error messages and exceptions are displayed verbosely and additional aids are given for effective development.
- The *Testing context* is used by FLOW3's test runner to realize a parallel FLOW3 instance acting as a sandbox for test cases.

Tip If FLOW3 throws some strange errors at you after you made code changes, make sure to either manually flush the cache or run the application in `Development` context - because caches are not flushed automatically in `Production` context.

The configuration for each context is located in directories of the same name:

Context Configurations

Directory	Description
<code>Configuration/</code>	Global configuration, for all contexts
<code>Configuration/Development/</code>	Configuration for the <code>Development</code> context
<code>Configuration/Production/</code>	Configuration for the <code>Production</code> context

2 Configuring FLOW3

FLOW3 should work fine with the default configuration delivered with the distribution. However, there are many switches you can adjust: use a different database engine, specify another location for logging, select a faster cache backend and much more. The easiest way to find out which options are available is taking a look at the default configuration of the FLOW3 package and other packages. The respective files are located in `Packages/Framework/packageKey/Configuration/`. Don't modify these files directly but rather copy the setting you'd like to change and insert it into a file within the global or context configuration directories.

FLOW3 uses the YAML format

Footnote [YAML Ain't Markup Language](http://yaml.org)<http://yaml.org>

for its configuration files. If you never edited a YAML file, you need to know that indenting has a special meaning and tabs are not allowed.

More detailed information about FLOW3's configuration management can be found in the [Reference Manual](#).

Note If you're running FLOW3 on a Windows machine, you do have to make some adjustments to the standard configuration because it will cause problems with long paths and filenames. By default FLOW3 caches files within the `Data/Temporary/Caches/` directory whose absolute path can eventually become too long for Windows (isn't that spooky?).

To avoid errors you should change the cache configuration so it points to a location with a very short absolute file path, for example `C:\tmp\`. Do that by adding the following line to the file `Configuration/FLOW3.yaml`:

```
utility: environment: temporaryDirectoryBase: 'C:/tmp/'
```

▼ Chapter 5: Modeling

Before we kickstart our first application, let's have a quick look in what FLOW3 differs from other frameworks.

We claim that FLOW3 *lets you concentrate on the essential* and in fact this is one major design goal we followed in the making of FLOW3. There are many factors which can distract developers from their principal task to create an application solving real-world problems. Most of them are infrastructure-related and reappear in almost every project: security, database, validation, persistence, logging, visualization and much more. FLOW3 preaches legible code, well-proven design patterns, true object orientation and provides first class support for Domain-Driven Design. And it takes care of most of the cross-cutting concerns, separating them from the business logic of the application.

Footnote http://en.wikipedia.org/wiki/Domain-driven_design

Footnote Note that we don't use these techniques for academic reasons. Personally I have never attended a lecture about software design – I just love clean code due to the advantages I discovered in my real-world projects.

▼ 1 Domain-Driven Design

Every software aims to solve problems within its subject area – its domain – for its users. All the product's other functions are just padding which serves to further this aim. If the domain of your software is the booking of hotel rooms, the reservation and cancellation of rooms are two of your main tasks. However, the presentation of booking forms or the logging of security-relevant occurrences do not belong to the domain 'hotel room bookings' and primarily serve to support the main task.

Most of the time it is easy to check whether a function belongs to a domain: imagine that you are booking a room from a receptionist. He is capable of accomplishing the task and will readily meet your request. Now imagine how

this employee would react if you asked him to render a booking form or to cache requests. These tasks fall outside his domain. Only in the rarest cases is the domain of an application 'software'. Rather most programs offer solutions for real life processes.

To master the complexity of your application it is therefore essential to neatly separate areas which concern the domain from the code which merely serves the infrastructure. For this you will need a layered architecture – an approach that has worked for decades. Even if you have not previously divided code into layers consciously, the mantra 'model view controller' should fall easily from your lips

Footnote If it doesn't, we recommend reading our introductory sections about MVC in the [FLOW3 reference](#).

. For the model which is part of this MVC pattern is at best a model of part of a domain. As a *domain model* it is separated from the other applications and resides in its own layer, the *domain layer*.

Tip Of course there is much more to say about Domain-Driven Design which doesn't have place in this tutorial. A good starter is the [section about principles](#) on the FLOW3 website.

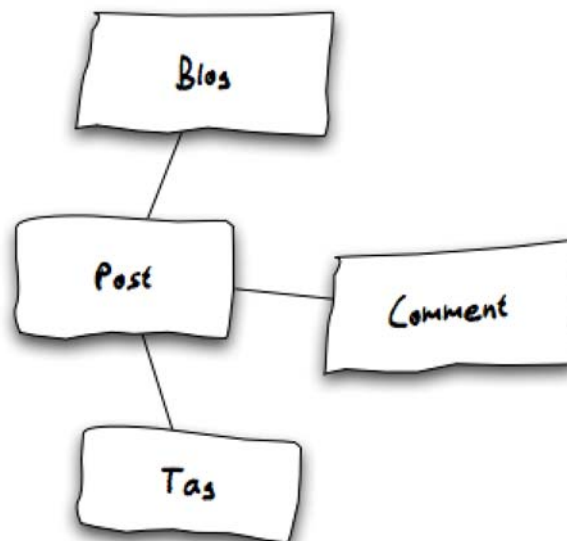
2 Domain Model

Our first FLOW3 application will be a blog system. Not because programming blogs is particularly fancy but because you will a) feel instantly at home with the domain and b) it is comparable with tutorials you might know from other frameworks.

So, how does our model look like? First we have blogs – our system is capable of managing multiple of those. Each blog has a number of posts, written by a certain author, with a title, publishing date and the actual post content. Each post can be tagged with an arbitrary number of tags. Finally, visitors of the blog may comment blog posts.

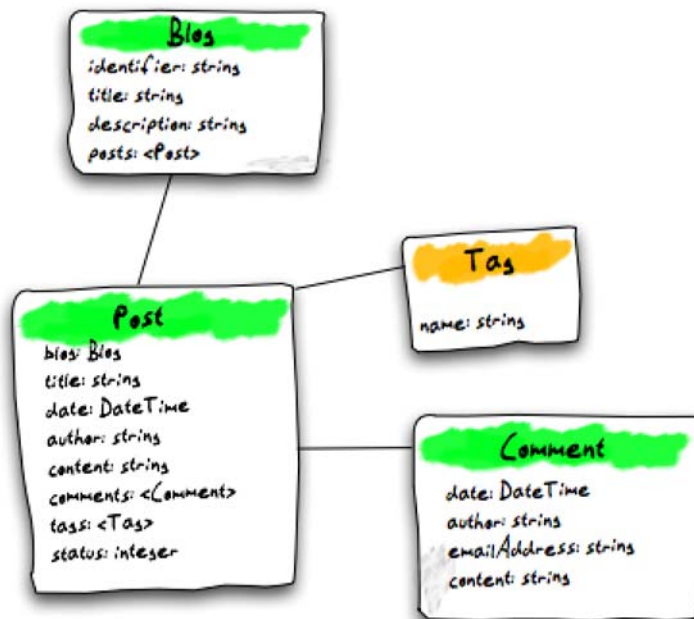
A first sketch shows which domain models (classes) we will need:

▼ *A simple model*



Let's add some properties to each of the models:

▼ *Domain Model with properties*



To be honest, the above model is not the best example of a rich Domain Model because – in contrast to Active Records

Footnote see http://en.wikipedia.org/wiki/Active_record_pattern

– those usually contain not only properties but also methods. For simplicity we also defined properties like `author` as simple strings – you'd rather plan in a dedicated `Author` object in a real-world model.

3 Repositories

Now that you have the models (conceptually) in place, you need to think about how you will access them. One thing you'll do is implementing a getter and setter method for each property you want to be accessible from the outside. You'll end up with a lot of methods like `getTitle`, `setAuthor`, `addComment` and the like

Footnote Of course we considered magic getters and setters. But then, how do you restrict read or write access to single properties? Furthermore, magic methods are notably slower and you lose the benefit of your IDE's autocompletion feature.

. Posts (i.e. `Post` objects) are stored in a `Blog` object in an array or better in an `SPIObjectStorage`

Footnote see <http://php.net/manual/en/class.splobjectstorage.php>

. For retrieving all posts from a given `Blog` all you need to do is calling the `getPosts` method of the `Blog` in question:

```
$posts = $blog->getPosts();
```

Executing `getComments` on the `Post` would return all related comments:

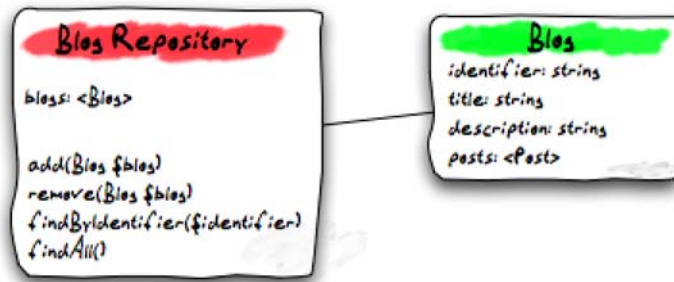
```
$comments = $post->getComments();
```

In the same manner `getTags` returns all tags attached to a given `Post`. But how do you retrieve a list of existing `Blogs`?

All objects which can't be found by another object need to be stored in a repository. In FLOW3 each repository is responsible for exactly one kind of an object (i.e. one class). Let's look at the relation between the

BlogRepository and the Blog:

▼ Blog Repository and Blog

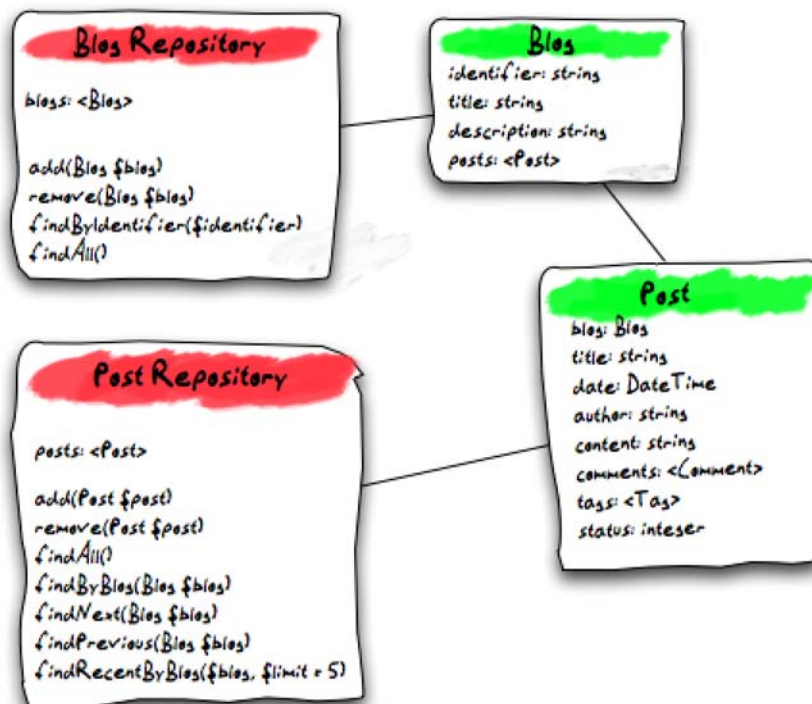


As you see, the Blog Repository provides methods for adding, removing and finding blogs.

Now, what if you want to display a list of the 5 latests posts, no matter what blog they belong to? One option would be to find all blogs, iterate over their posts and inspect each `date` property to create a list of the 5 most recent posts. Sounds slow? It is.

A much better way to find objects by a given criteria is querying a competent repository. Therefore, if you want to display a list of the 5 latest posts, you better create a dedicated `PostRepository` which provides a specialized `findRecentByBlog` method:

▼ A dedicated Post Repository



I silently added the `findPrevious` and `findNext` methods because you will later need them for navigating between posts.

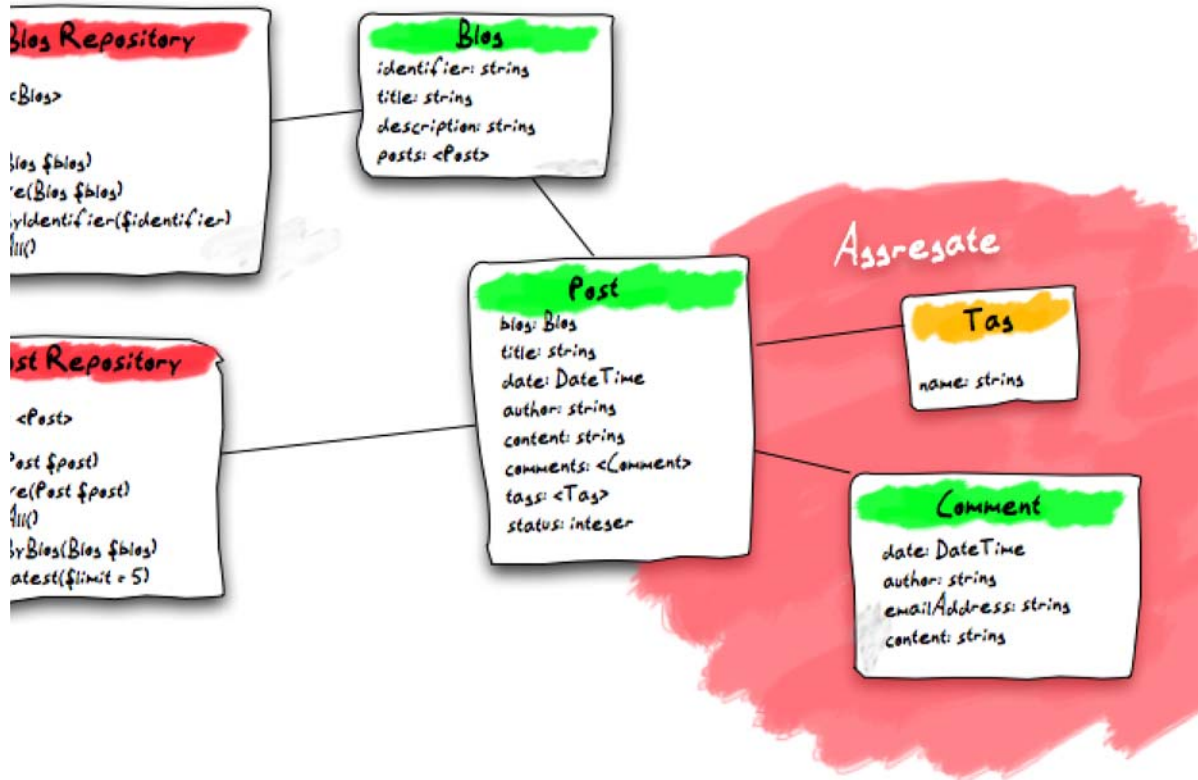
▼ 4 Aggregates

With the `PostRepository` you're now able to find posts independently from the `Blog`. There's no strict rule for when a

model requires its own repository. If you want to display comments independently from their posts and blogs, you'd surely need a Comment Repository, too. In this sample application you can do without it and find the comments you need by calling a getter method on the Post.

All objects which can only be found through a foreign repository, form an Aggregate. The object having its own repository (in this case `Post`) becomes the *Aggregate Root*:

▼ *The Post Aggregate*



The concept of aggregates simplifies the overall model because all objects of an aggregate can be seen as a whole: on deleting a post, the framework also deletes all associated comments and tags because it knows that no direct references from outside the aggregate may exist.

Enough for the modeling part. You'll surely want some more classes later but first let's get our hands dirty and start with the actual implementation!

▼ Chapter 6: Kickstart

FLOW3 makes it easy to start with a new application. The `Kickstart` package provides template based scaffolding for generating an initial layout of packages, controllers, models and views.

Note At the time of this writing these functions are only available through FLOW3's command line interface. Please note that this might change in the future because the philosophy of FLOW3 is using

- the command line for *automatization* and *system administrative tasks* and
- a clear web interface for *modeling* and *development*

▼ 1 Command Line Tool

The script `flow3` resides in the main directory of the FLOW3 distribution. From a Unix shell you should be able to run the script by entering `./flow3`:

```
myhost:tutorial johndoe$ ./flow3
FLOW3 Command Line Interface

usage: flow3 <options> <command>

Options:

-h, --help          - print this message
-p, --production    - execute in production context

Available commands:

package list available          - list available packages
package list active            - list active packages
package create <package-key>   - create a new package
package activate <package-key> - activate a package
package deactivate <package-key> - deactivate a package
package delete <package-key>   - delete a package

cache flush                    - flush all caches

testing <package-key> <output-directory> [<testcase> [<coverage-directory>]]
                                - run unit tests
                                <package-key> Package to test (mandatory)
                                <output-directory> path to write the logfile.xml
                                <testcase> only run this testcase (optional)
                                <coverage-directory> path to write the clover.xml

kickstart package <package-key>
                                - kickstart a new package, including a standard cont
                                <package-key> Package for the controller (mandator

kickstart controller <package-key> [<controller-name>]
                                - generate a controller
                                <package-key> Package for the controller (mandator
                                <controller-name> Name of the controller (optional
```

Depending on your FLOW3 version you'll see more or less the above available command listed.

Note We haven't developed a Windows batch script yet so for the time being you'll have to call FLOW3 manually. Before you can run the FLOW3 command line script you need to set some environment variables:

```
c:\> set FLOW3_CONTEXT=Development
c:\> set FLOW3_ROOTPATH=C:\xampp\htdocs\tutorial
```

```
c:\> set FLOW3_WEBPATH=C:\xampp\htdocs\tutorial
```

Listing the available packages is then as easy as typing:

```
c:\> (php Packages\Framework\FLOW3\Scripts\FLOW3.php  
      FLOW3 Package Manager listavailable)
```

▼ 2 Kickstart the package

Let's create a new package *Blog*:

```
myhost:tutorial johndoe$ ./flow3 kickstart package Blog
```

or on Windows:

```
c:\xampp\htdocs\tutorial> (php Packages\Framework\FLOW3\Scripts\FLOW3.php  
Kickstart Kickstart generatePackage --packageKey Blog)
```

The kickstarter will create two files

```
+ Packages/Application/Blog/Classes/Controller/StandardController.php  
+ Packages/Application/Blog/Resources/Private/Templates/Standard/index.html
```

and the directory `Packages/Application/Blog/` should now contain the skeleton of the future `Blog` package:

```
myhost:tutorial johndoe$ cd Packages/Application/  
myhost:Application johndoe$ find Blog  
Blog  
Blog/Classes  
Blog/Classes/Controller  
Blog/Classes/Controller/StandardController.php  
Blog/Configuration  
Blog/Documentation  
Blog/Meta  
Blog/Meta/Package.xml  
Blog/Resources  
Blog/Resources/Private  
Blog/Resources/Private/Templates  
Blog/Resources/Private/Templates/Standard  
Blog/Resources/Private/Templates/Standard/index.html  
Blog/Tests
```

Switch to your web browser and check if the generated controller produces some output:

eated Fluid template!

oller:



▼ 3 Kickstart Controllers

If you look at the drawing of our overall model you'll notice that you need controllers for the most important domain models, being `Blog`, `Post` and `Comment`. Create them with the kickstarter as well:

```
myhost:tutorial johndoe$ ./flow3 kickstart controller Blog Blog,Post,Comment
```

or on Windows:

```
c:\xampp\htdocs\tutorial> (php Packages\Framework\FLOW3\Scripts\FLOW3.php
Kickstart Kickstart generateController --packageKey Blog --controllerName
Blog,Post,Comment)
```

resulting in:

```
+ Packages/Application/Blog/Classes/Controller/BlogController.php
+ Packages/Application/Blog/Resources/Private/Templates/Blog/index.html
+ Packages/Application/Blog/Classes/Controller/PostController.php
+ Packages/Application/Blog/Resources/Private/Templates/Post/index.html
+ Packages/Application/Blog/Classes/Controller/CommentController.php
+ Packages/Application/Blog/Resources/Private/Templates/Comment/index.html
```

These new controllers can now be accessed via `http://dev.tutorial.local/blog/blog`, `http://dev.tutorial.local/blog/post` and `http://dev.tutorial.local/blog/comment` respectively.

Tip If you can't access the newly created controllers one reason might be that you did not run FLOW3 in the development context (did you set the `FLOW3_CONTEXT` environment variable as explained earlier?). As already mentioned, FLOW3 does not clear caches automatically in a production context so you better work in development mode while you're developing.

Please delete the file `StandardController.php` and its corresponding template directory as you won't need them for our sample application.

▼ 4 Kickstart Models and Repositories

The kickstarter can also generate models and repositories

```
Footnote Want to try it out? The syntax is ./flow3 kickstart model PackageKey ModelName
propertyName:type propertyName:type ... or on Windows php
Packages\Framework\FLOW3\Scripts\FLOW3.php Kickstart Kickstart generateModel
--packageKey Blog --modelName ModelName foo:string bar:integer
```

. However, at this point you will stop using the kickstarter because a) writing models and repositories by hand is really easy and b) as mentioned before, the command line won't be the preferred way of generating scaffolds in the future. We are not completely happy with the parameter syntax yet and therefore better don't teach it to you ...

▼ Chapter 7: Model and Repository

Usually this would now be the time to write a database schema which contains table definitions and lays out relations between the different tables. But FLOW3 doesn't deal with tables. You won't even access a database manually nor will you write SQL. The very best is if you completely forget about tables and databases and think only in terms of objects.

Code Examples

The following sections contain a lot of code which we'll go through step by step. You may but don't have to copy and paste the code to follow the examples.

If you're lost or just like to peek at the final code, go to the resources folder of the `GettingStarted` package: In `Private/CheatSheet/` you'll find all files mentioned in this tutorial (in fact `CheatSheet` contains the whole package `Blog` which you'll develop yourself step by step).

Domain models are really the heart of your application and therefore it is vital that this layer stays clean and legible. In a FLOW3 application a model is just a plain old PHP object

```
Footnote We love to call them POPOs, similar to POJOs http://en.wikipedia.org/wiki/Plain\_Old\_Java\_Object
```

. There's no need to write a schema definition, subclass a special base model or implement a required interface. All FLOW3 requires from you as a specification for a model is a proper documented PHP class containing properties.

Before you continue first create the directory for your domain models:

```
myhost:tutorial johndoe$ mkdir -p Packages/Application/Blog/Classes/Domain/Model
```

The directory structure and filenames follow the conventions of our [Coding Guidelines](#) which basically means that the directories reflect the classes' namespace while the filename is identical to the class name.

Tip Namespaces have been introduced in PHP 5.3. If you're unfamiliar with its funny backslash syntax you might want to have a look at the [PHP manual](#).

▼ 1 Blog Model

The code for your `Blog` model (`.../Blog/Classes/Domain/Model/Blog.php`) might look like the following:

```
<?php
declare(ENCODING = 'utf-8');
namespace F3\Blog\Domain\Model;
```

```

/**
 * A blog
 *
 * @scope prototype
 * @entity
 */
class Blog {

    /**
     * The blog's identifier.
     *
     * @var string
     * @identity
     */
    protected $identifier = '';

    /**
     * The blog's title.
     *
     * @var string
     */
    protected $title = '';

    /**
     * A short description of the blog
     *
     * @var string
     */
    protected $description = '';

    /**
     * The posts contained in this blog
     *
     * @var \SplObjectStorage<\F3\Blog\Domain\Model\Post>
     */
    protected $posts;

    /**
     * Constructs a new Blog
     *
     */
    public function __construct() {
        $this->posts = new \SplObjectStorage();
    }
}

```

```

/**
 * Sets this blog's identifier
 *
 * @param string $identifier The identifier
 * @return void
 */
public function setIdentifier($identifier) {
    $this->identifier = $identifier;
}

/**
 * Returns the blog's identifier
 *
 * @return string The blog's identifier
 */
public function getIdentifier() {
    return $this->identifier;
}

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

/**
 * Returns the blog's title
 *
 * @return string The blog's title
 */
public function getTitle() {
    return $this->title;
}

/**
 * Sets the description for the blog
 *
 * @param string $description The blog description or "tag line"
 * @return void
 */
public function setDescription($description) {
    $this->description = $description;
}

```

```

}

/**
 * Returns the description
 *
 * @return string The blog description
 */
public function getDescription() {
    return $this->description;
}

/**
 * Adds a post to this blog
 *
 * @param \F3\Blog\Domain\Model\Post $post
 * @return void
 */
public function addPost(\F3\Blog\Domain\Model\Post $post) {
    $post->setBlog($this);
    $this->posts->attach($post);
}

/**
 * Returns all posts in this blog
 *
 * @return \SplObjectStorage<\F3\Blog\Domain\Model\Post> The posts of this blog
 */
public function getPosts() {
    return clone $this->posts;
}
}
?>

```

As you can see there's nothing really fancy in it, the class mostly consists of getters and setters. Let's take a closer look at the model line-by-line:

```
declare(ENCODING = 'utf-8');
```

This declaration will help PHP 6 to determine the encoding of the PHP file - we already use it for being upwards compatible

Footnote In case our code survives the release date of PHP 6, probably scheduled for 2025.

```
namespace F3\Blog\Domain\Model;
```

This namespace declaration must be the very first code in your file - aside from the encoding declaration.

```
/**
 * A blog
 *
 * @scope prototype
 * @entity
 */
class Blog {
```

On the first glance this looks like a regular comment block, but it's not. This comment contains two *annotations* which are an important building block in FLOW3's configuration mechanism.

The `@scope` annotation defines the object scope. By default only one global instance exists of each class - this is called the *singleton scope*. If we want to allow multiple instances at a time (and we do want multiple `Blog` objects) we need to annotate the class with `@scope prototype`. Don't worry about this now, you'll soon learn more about scopes and the object management in general.

The second annotation marks this class as an `@entity`. This is an important piece of information for the persistence framework because it declares that

- this model is an *entity* according to the concepts of Domain-Driven Design
- instances of this class can be persisted

According to DDD, an entity is an object which has an identity, i.e. even if two objects with the same values exist, their identity matters.

The model's properties are implemented as regular class properties:

```
/**
 * The blog's identifier.
 *
 * @var string
 * @identity
 */
protected $identifier = '';

/**
 * The blog's title.
 *
 * @var string
 */
protected $title = '';

/**
 * A short description of the blog
 *
 * @var string
 */
protected $description = '';

/**
```

```

* The posts contained in this blog
*
* @var \SplObjectStorage<\F3\Blog\Domain\Model\Post>
*/
protected $posts;

```

Please note that the property `$identifier` is not a convention or requirement by FLOW3. Although the persistence framework stores some unique identifier for each object internally, it doesn't require a property for this purpose in the model. The reason why the `Blog` model contains an `$identifier` property lies in the domain of the blog. The property could equally been named `$blogName` or the like. Independent thereof the `@identity` annotation tells FLOW3 that this property has the function of an identifier in your domain which mainly serves as a source of information for building nice looking URLs.

Each property comes with a `@var` annotation which declares its type. Any type is fine, be it simple types like `string`, `integer`, `boolean` or classes like `\DateTime`, `\F3\Foo\Domain\Model\Bar` or `\ArrayObject`. Regarding the type, the `@var` annotation of the `$posts` property differs a bit from the remaining comments. This property holds a list of `Post` objects contained by this blog - in fact this could easily have been an array:

```

/**
 * The posts contained in this blog
 *
 * @var array<\F3\Blog\Domain\Model\Post>
 */
protected $posts = array();

```

However, an array would allow `$posts` to contain the same post multiple times. We therefore use an `SplObjectStorage` which guarantees the uniqueness of each post attached to it.

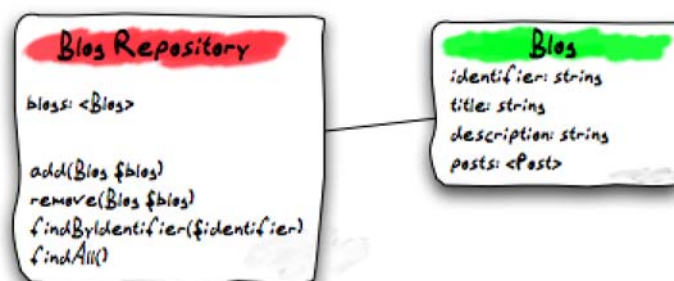
The class name bracketed by the less-than and greater-than signs gives an important hint on the content of the array or object storage. There are a few situations in which FLOW3 relies on this information.

The remaining code shouldn't hold any surprises - it only serves for setting and retrieving the blog's properties. This again, is no requirement by FLOW3 - if you don't want to expose your properties it's fine to not define any setters or getters at all. The persistence framework uses other ways to access the properties' values ...

▼ 2 Blog Repository

According to our earlier reasonings, you need a repository for storing blogs:

▼ Blog Repository and Blog



A repository acts as the bridge between the holy lands of business logic (domain models) and the dirty underground of infrastructure (data storage). This is the only place where queries to the persistence framework

take place - you never want to have those in your domain models.

Implementing a vanilla repository for blogs is as easy as this

(.../Blog/Classes/Domain/Repository/BlogRepository.php):

```
<?php
declare(ENCODING = 'utf-8');
namespace F3\Blog\Domain\Repository;

/**
 * A repository for Blogs
 */
class BlogRepository extends \F3\FLOW3\Persistence\Repository {
}
?>
```

As you see there's no code you need to write for the standard cases because the base repository already comes with methods like `add`, `remove`, `findAll`, `findBy*` and `findOneBy*`

Footnote `findBy*` and `findOneBy*` are magic methods provided by the base repository which allow you to find objects by properties. The `BlogRepository` for example would allow you to call magic methods like `findByDescription('foo')` or `findOneByTitle('bar')`.

methods.

Remember that a repository can only store one kind of an object, in this case blogs. The type is derived from the repository name: because you named this repository `BlogRepository` FLOW3 assumes that it's supposed to store `Blog` objects.

▼ Chapter 8: Controller

Now that you have the first model and repository in place you can move forward to creating a blog, storing it in the repository and retrieving it again.

▼ 1 Basic Blog Controller

The kickstarter created a very basic blog controller containing only one action, the `indexAction`. For demonstrating the interaction between the controller, model and repository you modify the class as follows:

```
<?php
declare(ENCODING = 'utf-8');
namespace F3\Blog\Controller;

// ...

class BlogController extends \F3\FLOW3\MVC\Controller\ActionController {

    /**
     * @var \F3\Blog\Domain\Repository\BlogRepository
     * @inject
     */
```

```

protected $blogRepository;

/**
 * Index action
 *
 * @return string HTML code
 */
public function indexAction() {
    $blogs = $this->blogRepository->findAll();
    $output = '<ul>';
    foreach ($blogs as $blog) {
        $output .= '<li>' . $blog->getTitle() . '</li>';
    }
    $output .= '</ul>';
    return $output;
}

/**
 * Create action
 *
 * @return void
 */
public function createAction() {
    $blog = $this->objectFactory->create('F3\Blog\Domain\Model\Blog');
    $blog->setIdentifier('fooblog');
    $blog->setTitle('My first blog');

    $this->blogRepository->add($blog);
    $this->redirect('index');
}
}
?>

```

The `indexAction` now retrieves all available blogs from the `BlogRepository` and outputs the title in an unordered list

Footnote Don't worry, the action won't stay like this - of course we'll later move all HTML rendering code to a dedicated view.

. The line

```
$blogs = $this->blogRepository->findAll();
```

should be pretty self-explanatory. But where do you get the `blogRepository` from?

```

/**
 * @var \F3\Blog\Domain\Repository\BlogRepository
 * @inject

```

```
*/
protected $blogRepository;
```

The property declaration for `$blogRepository` is marked with an `@inject` annotation. This signals to the object framework: I need the blog repository here, please make sure it's stored in this member variable. In effect FLOW3 will inject the blog repository into the `$blogRepository` property right after your controller has been instantiated. And because the blog repository's scope is `singleton`

Footnote Remember, `singleton` is the default object scope and because the `BlogRepository` does not contain a `@scope` annotation, it has the default scope.

, the framework will always inject the same instance of the repository.

There's a lot more to discover about *Dependency Injection* and we recommend that you read the whole chapter about objects in the [FLOW3 Reference](#) once you start with your own coding.

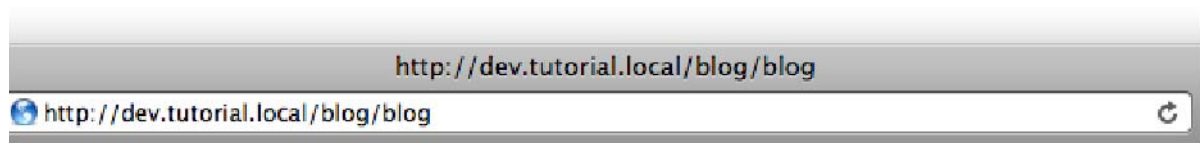
Fine, the `indexAction` can now display blogs. A quick look at `http://dev.tutorial.local/blog/blog`

Footnote The first `blog` stands for the package `Blog` and the second one specifies the controller `BlogController`.

reveals ... a blank page. However you should find the empty unordered list if you look at the page's source code.

Create a blog object by accessing `http://dev.tutorial.local/blog/blog/create`. If all went right, you should see the following output:

▼ *Output of the indexAction*



As you see, the `createAction` successfully created a blog and added it to the repository. Had you omitted the line

```
$this->blogRepository->add($blog);
```

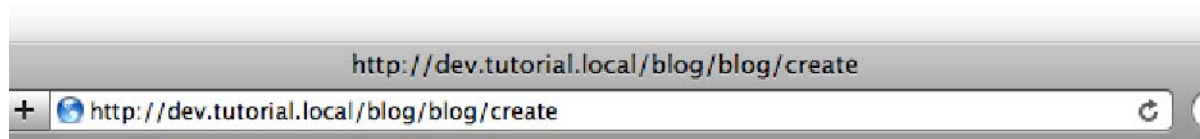
the blog object would have been created but not persisted – the list rendered by the `indexAction` would be empty.

▼ 2 Create Action

The create action served well for demonstrating how a new blog can be created, but as it is now it doesn't help us in real life. Instead of creating the same blog again and again it should receive the blog to be created as an argument. Check out this new `createAction`:

```
/**
 * Creates a blog
 *
 * @param F3\Blog\Domain\Model\Blog $newBlog A fresh Blog object which has not ←
 *       yet been added to the repository
 * @return void
 */
public function createAction(\F3\Blog\Domain\Model\Blog $newBlog) {
    $this->blogRepository->add($newBlog);
    $this->pushFlashMessage('Your new blog was created.');
```

This method expects a parameter `$newBlog` which is the `Blog` object to be persisted. The code is quite straight-forward: add the blog to the repository, add a message to some flash message stack and redirect to the index action. Try calling the `createAction` now by accessing `http://dev.tutorial.local/blog/blog/create` again:



ile trying to call F3\Blog\Controller\BlogController->createAction(). Error: Required property 'new



FLOW3 analyzed the new method signature and automatically registered `$newBlog` as a required argument for `createAction`. Because no such argument was passed to the action, the controller exits with an error.

So, how do you create a new blog? You need to create some HTML form which allows us to enter the blog details and then submits the information to the `createAction`. But you don't want the controller let rendering such a form – this is clearly a task for the view!

▼ Chapter 9: View

The view's responsibility is solely the visual presentation of data provided by the controller. In FLOW3 views are cleanly decoupled from the rest of the MVC framework. This allows you to either take advantage of Fluid (FLOW3's template engine), write your own custom PHP view class or use almost any other template engine by writing a thin wrapper building a bridge between FLOW3's interfaces and the template engine's functions. In this tutorial we focus on Fluid-based templates as this is what you usually want to use.

▼ 1 Resources

Before we design our first Fluid template we need to spend a thought on the resources our template is going to use (I'm talking about all the images, style sheets and javascript files which are referred to by your HTML code). You remember that only the `Web` directory is accessible from the web, right? And the resources are part of the package and thus hidden from the public. That's why FLOW3 comes with a (at least in the future) powerful resource manager

Footnote In the future? Yes, because until now we only managed to implement the absolute basics of it. Though we have great plans for what the resource manager will be able to do we have to admit that the current solution is a bit unpolished.

whose main task is to manage access to your package's resources.

The deal is this: All files which are located in the *public resources directory* of your package will automatically be mirrored to the public resources directory below the `Web` folder. Let's take a look at the directory layout of the `Blog` package:

▼ Directory structure of a FLOW3 package

Directory	Description
Classes/	All the .php class files of your package
Documentation/	The package's manual and other documentation
Meta/	Package.xml and other package meta information
Resources/	Top folder for resources
Resources/Public/	Public resources - will be mirrored to the <code>Web</code> directory
Resources/Private/	Private resources - won't be mirrored to the <code>Web</code> directory

No matter what files and directories you create below `Resources/Public/` - all of them will be copied to `/Web/Resources/Packages/Blog/` on the next hit.

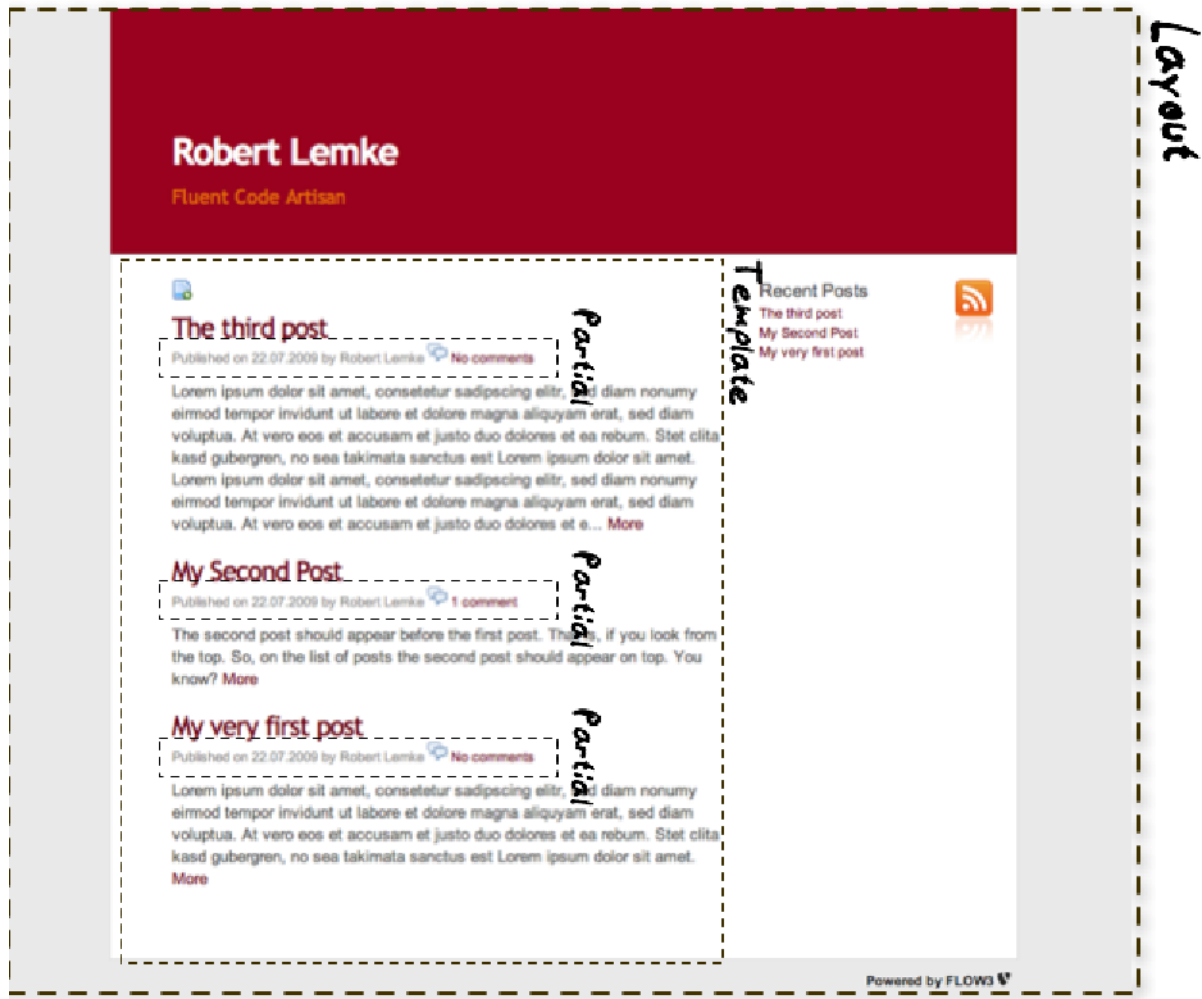
Tip There are more possible directories in a package and we do have some conventions for naming certain sub directories. All that is explained in fine detail in the [FLOW3 reference manual](#).

Important For the blog example in this tutorial we created some style sheets and icons. If you'd like to brush up the following examples a little, then it's now time to copy all files from `Packages/Application/GettingStarted/Resources/Private/CheatSheet/Resour` to your blog's public resources folder (`Packages/Application/Blog/Resources/Public`).

▼ 2 Layouts

Fluid knows the concepts of layouts, templates and partials. Usually all of them are just plain HTML files which contain special tags known by the Fluid template view. The following figure illustrates the use of layout, template and partials in our blog example:

▼ *Layout, Template and Partial*



A Fluid layout provides the basic layout of the output which is supposed to be shared by multiple templates. You will use the same layout throughout this tutorial - only the templates will change depending on the current controller and action. Elements shared by multiple templates can be extracted as a partial to assure consistency and avoid duplication.

Let's build a simple layout for your blog. You only need to create a new folder `Layouts` inside the `Blog/Resources/Private/` directory and save the following code in a file called `master.html`:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

```

<f:base />
<title>{blog.title}</title>
<link rel="stylesheet" href="{f:uri.resource('Blog.css')}}" type="text/css" />
</head>
<body>
<div id="header">
  <f:if condition="{blog}">
    <f:link.action action="index" controller="Post" arguments="{blog: blog}">
      <h1 class="title">{blog.title}</h1></f:link.action>
      <p class="description">{blog.description}</p>
    </f:if>
  </div>
<div id="maincontainer">
  <div id="mainbox"><f:render section="mainbox" /></div>
  <div class="clear"></div>
</div>
<div id="footer">
  <a href="http://flow3.typo3.org">Powered by FLOW3
    
  </a>
</div>
</body>
</html>

```

On first sight this looks like plain HTML code, but you'll surely notice the various `<f: ... >` tags. Fluid provides a range of view helpers which are addressed by these tags. By default they live in the `f` namespace resulting in tags like `<f:base />` or `<f:if>`. You can define your own namespaces and even develop your own view helpers, but for now let's look at what you used in your layout:

The first Fluid tag used is the `<f:base />` tag. This tag instructs Fluid to render an HTML `<base>` tag containing the correct absolute base URI for your site – in your case resulting in

```
<base href="http://dev.tutorial.local/"></base>
```

The second occurrence of Fluid markup is actually no tag but a variable accessor:

```
<title>{blog.title}</title>
```

As you will see in a minute, Fluid allows your controller to define variables for the template view. In case there is a *current* blog (e.g. while you're displaying a list of blog posts), you'll need to make sure that your controller assigns the current `Blog` object to the template variable `blog`. The value of such a variable can be inserted anywhere in your layout, template or partial by inserting the variable name wrapped by curly braces. However, in the above case `blog` is not a value you can output right away – it's an object. Fortunately Fluid can display properties of an object which are accessible through a getter function: to display the blog title, you just need to note down `{blog.title}`.

The third appearance of Fluid syntax is an alternative way to address view helpers, the view helper shorthand syntax.

Warning The short hand syntax is probably subject to change, we're at least not 100% sure about it yet

```
<link rel="stylesheet" href="{f:uri.resource('Blog.css')}" type="text/css" />
```

This instructs the URI view helper to create a relative resource URL pointing to your style sheet. The generated HTML code will look like this:

```
<link rel="stylesheet" href="Resources/Packages/Blog/Blog.css" type="text/css" />
```

If you look at the remaining markup of the layout you'll find more uses of view helpers, including conditions and link generation. There's only one more view helper you need to know about before proceeding with our first template, the *render* view helper:

```
<f:render section="mainbox" />
```

This tag tells Fluid to insert the section `mainbox` defined in the current template at this place. For this to work there must be a section with the specified name in the template referring to the layout – because that's the way it works: A template declares on which layout it is based on, defines sections which in return are included by the layout. Confusing? Let's look at a concrete example.

▼ 3 Templates

Templates are, as already mentioned, tailored to a specific action. The action controller chooses the right template automatically according to the current package, controller and action - if you follow the naming conventions. Let's replace the automatically generated template for the Blog controller's index action in `Blog/Resources/Private/Templates/Blog/index.html` by some more meaningful HTML:

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

<f:section name="mainbox">
  <h1 class="flow3-firstHeader">Getting Started - Blog Example</h1>

  <f:for each="{flashMessages}" as="message">
    <p class="bodytext"><strong>{message}</strong></p>
  </f:for>

  <f:if condition="{blogs}">
    <f:then>
      <p class="bodytext">Here is a list of blogs:</p>
      <ul>
        <f:for each="{blogs}" as="blog">
          <li>
            <h3>
              <f:link.action action="index" controller="Post" arguments="{blog : blog}">
                {blog.title}
              </f:link.action>
            </h3>
            <p>
              <f:format.nl2br>{blog.description}</f:format.nl2br>
              <f:link.action action="edit" controller="Blog" arguments="{blog : blog}">
                Edit</f:link.action>
              <f:link.action action="delete" controller="Blog" arguments="{blog : blog}">
                Delete</f:link.action>
            </p>
          </li>
        </f:for>
      </ul>
    </f:then>
  </f:if>
</f:section>
```

```

    </p>
  </li>
</f:for>
</ul>
<p><f:link.action action="new">Create another blog</f:link.action></p>
</f:then>
<f:else>
  <p><strong>
    <f:link.action action="new">Create your first blog</f:link.action>
  </strong></p>
</f:else>
</f:if>
</f:section>

```

There you have it: In the first line of your template there's a reference to the master layout. All HTML code is wrapped in a `<f:section>` tag. Even though this is the way you usually want to design templates, you should know that using layouts is not mandatory – you could equally put all your code into one template and omit the `<f:layout>` and `<f:section>` tags.

Take a quick look at the template. You'll note that we're using a new view helper right at the top – a `for` loop which iterates over `flashMessages`. Well, maybe you remember this line you put into the `createAction` of our `BlogController`:

```
$this->pushFlashMessage('Your new blog was created.');
```

Flash messages are a great way to display success or error messages to the user. And because they are so useful, the action controller assigns the current flash messages automatically to the template variable `flashMessages`. Therefore, if you create a new blog, you'll see the message "Your new blog was created" at the top of your blog index on the next hit.

The main job of this template is to display a list of existing blogs, let you edit or delete them or create new ones. An `<f:if>` condition makes sure that the list of blogs is only rendered if `blogs` actually contains blogs. But currently the view doesn't know anything about blogs – you need to adapt the `indexAction` of the `BlogController` to assign blogs to the view:

```

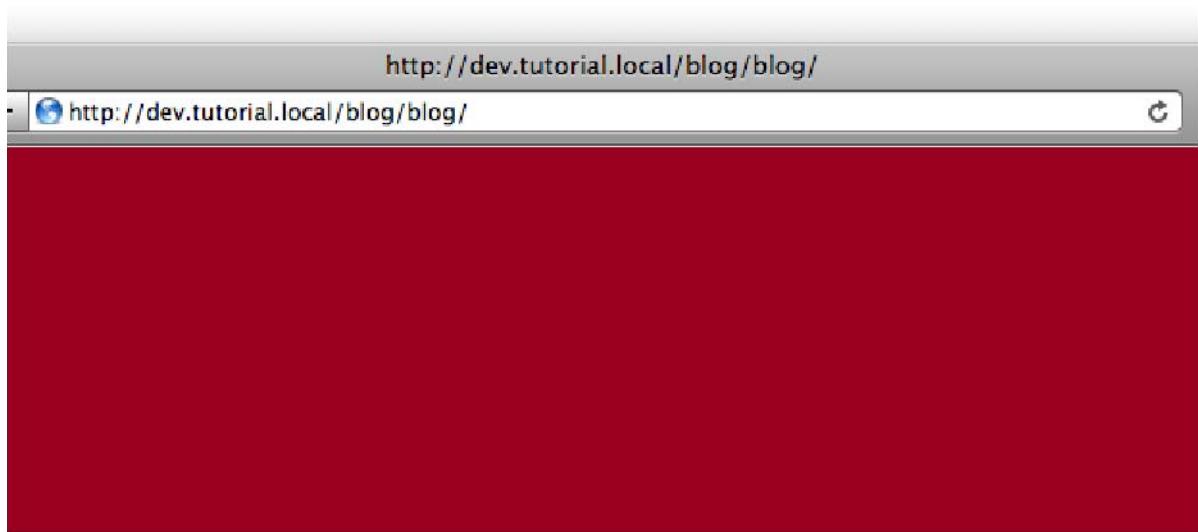
/**
 * Index action
 *
 * @return void
 */
public function indexAction() {
  $blogs = $this->blogRepository->findAll();
  $this->view->assign('blogs', $blogs);
}

```

To fully understand the above code you need to know two facts:

1. `$this->view` is automatically set by the action controller and points to a Fluid template view.
2. if an action method returns `NULL`, the controller will automatically call `$this->view->render()` after executing the action.

You should now see a list of available blogs by accessing `http://dev.tutorial.local/blog/blog`:



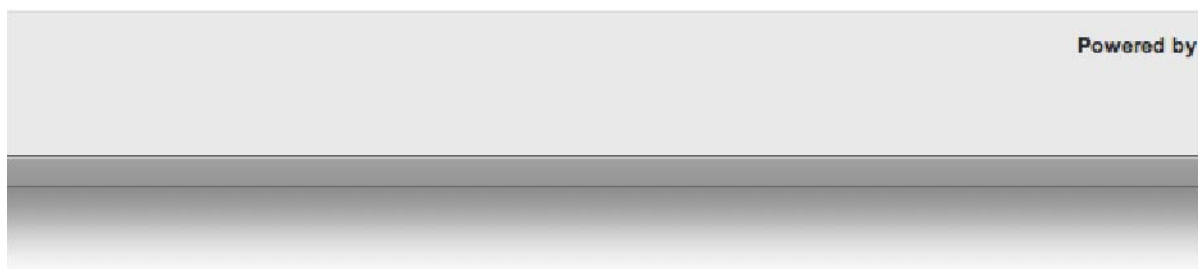
Getting Started - Blog Example

Here is a list of blogs:

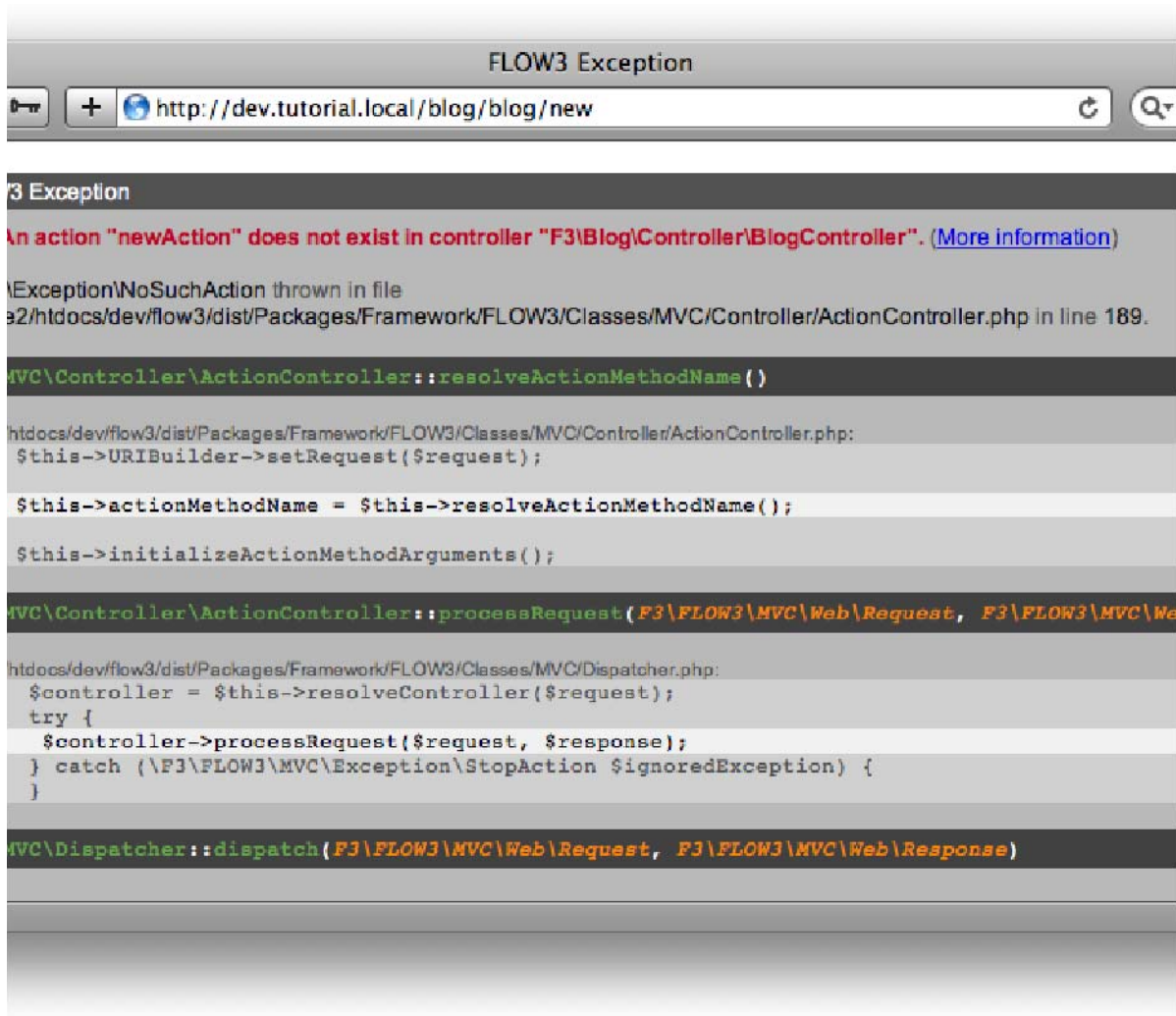
My first blog

Edit Delete

Create another blog



Creating a new blog won't work yet because, you didn't implement a `newAction`:



4 Forms

4.1 Create a New Blog

Time to create a form which allows you to enter details for a new blog. The first component you need is the `newAction` whose sole purpose is displaying the form:

```
/**  
 * New action  
 *  
 * @return void  
 */  
public function newAction() {  
}
```

No code? No code. What will happen is this: the action controller selects the `new.html` template and assigns it to `$this->view` which will automatically be rendered after `newAction` has been called. That's enough for displaying the form.

The second component is the actual form. Create a new template `new.html` in the `Templates/Blog/` folder:

```

<f:layout name="master" />

<f:section name="mainbox">
  <h2 class="flow3-firstHeader">Create a new blog</h2>
  <p>Enter information about your new blog below:</p>
  <f:form method="post" action="create" name="newBlog">
    <label for="identifier">
      Identifier <span class="required">(required)</span></label>
    </label><br />
    <f:form.textbox property="identifier" id="identifier" />
    <br />
    <label for="name">Title <span class="required">(required)</span></label><br />
    <f:form.textbox property="title" id="title" />
    <br />
    <label for="description">Description</label><br />
    <f:form.textarea property="description" rows="2" cols="40" id="description" />
    <br />
    <f:form.submit value="Create blog" />
  </f:form>
</f:section>

```

Here is how it works: The `<f:form>` view helper renders a form tag. Its attributes are similar to the action link view helper you might have seen in previous examples: `action` specifies the action to be called on submission of the form, `controller` would specify the controller and `package` the package respectively. If `controller` or `package` are not set, the URI builder will assume the current controller or package respectively. `name` finally declares the name of the form and at the same time specifies *the name of the action method argument* which will receive the form values.

It is important to know that the whole form is (usually) bound to one object and that the values of the form's elements become property values of this object. In this example the form contains (property) values for a blog object. The form's elements are named after the class properties of the `Blog` domain model: `identifier`, `name` and `description`. Let's look at the `createAction` again:

```

/**
 * Creates a new blog
 *
 * @param F3\Blog\Domain\Model\Blog $newBlog A fresh Blog object which has not ←
 *       yet been added to the repository
 * @return void
 */
public function createAction(\F3\Blog\Domain\Model\Blog $newBlog) {
    $this->blogRepository->add($newBlog);
    $this->pushFlashMessage('Your new blog was created.');
```

It's important that the `createAction` uses the type hint `\F3\Blog\Domain\Model\Blog` and comes with a proper `@param` annotation because this is how FLOW3 determines the type to which the submitted form values must be

converted. Because this action requires a `Blog` it gets a `blog` (object) - as long as the property names of the object and the form match.

Time to test your new `newAction` and its template - click on the *Create another blog* link which lets the `newAction` render this form:

Create a new blog

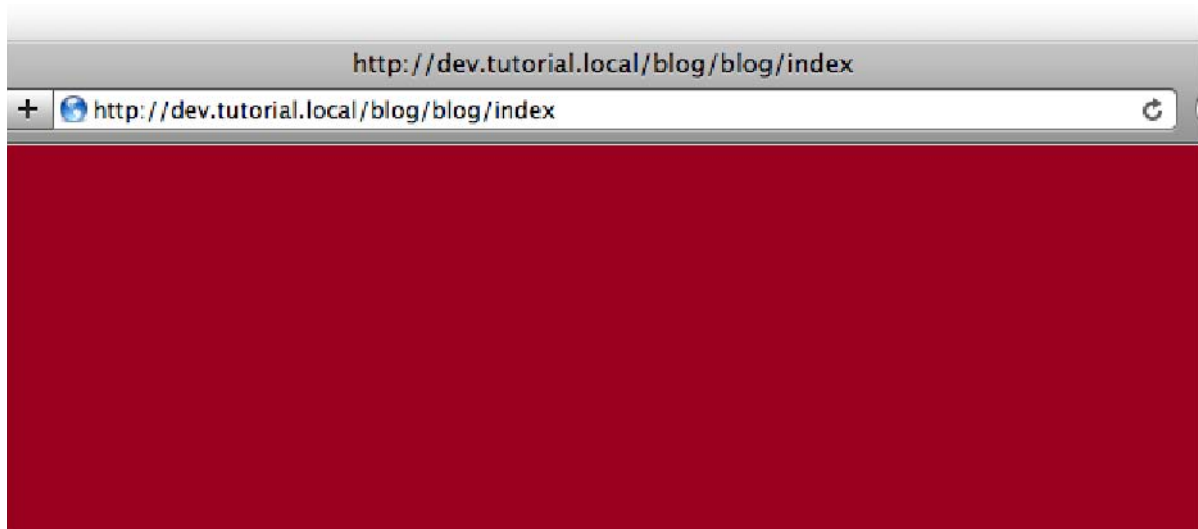
Enter information about your new blog below:

identifier (required)

Title (required)

Description

Enter some data and click the *Create blog* button:



Getting Started - Blog Example

Your new blog was created.

Here is a list of blogs:

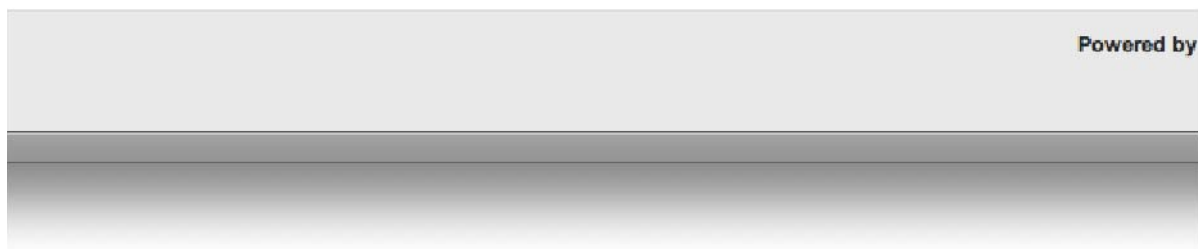
Robert Lemke

[View](#) [Code](#) [Artisan](#) [Edit](#) [Delete](#)

My first blog

[View](#) [Edit](#) [Delete](#)

[Create another blog](#)



You should now find your new blog in the list of blogs.

4.2 Edit a Blog

While you're dealing with forms you should also create a form for editing an existing blog. The `editAction` will display this form.

In order to display the current values of the blog to be edited you need to pass it the blog object:

```

/**
 * Edits an existing blog
 *
 * @param \F3\Blog\Domain\Model\Blog $blog The blog to be edited.
 * @return string Form for editing the existing blog
 */
public function editAction(\F3\Blog\Domain\Model\Blog $blog) {
    $this->view->assign('blog', $blog);
}

```

This is pretty straight forward: the link you rendered in the `index.html` template passes an argument `$blog` to your edit action and the action on its part assigns the blog to the template.

Create the new template `Templates/Blog/edit.html` and insert the following HTML code:

```

<f:layout name="master" />

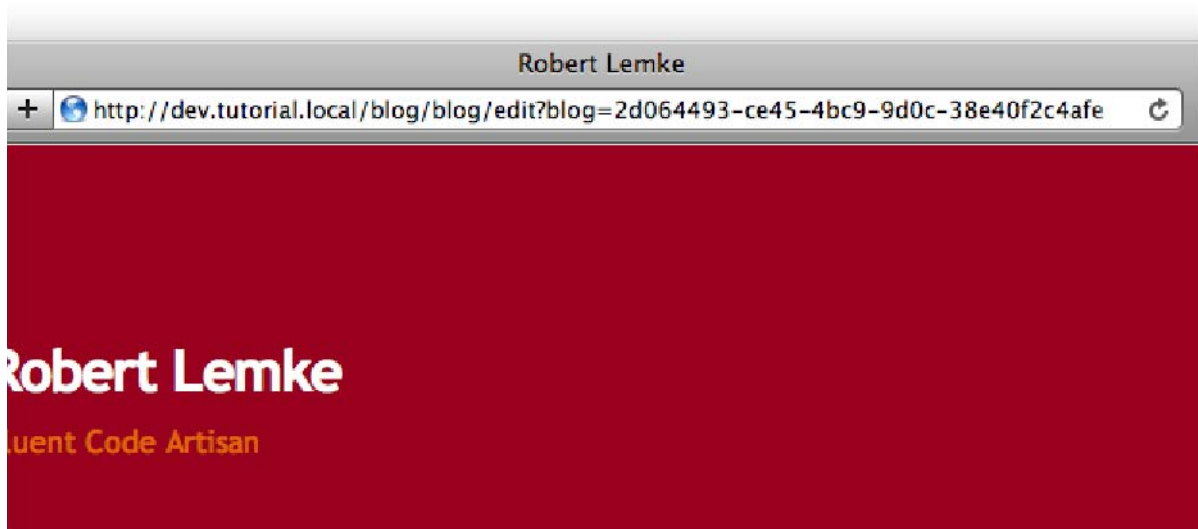
<f:section name="mainbox">
    <div class="flow3-header flow3-header-n1">
        <h1 class="flow3-firstHeader">Edit blog "{blog.identifier}"</h1>
    </div>
    <p class="bodytext">Update the information below:</p>
    <f:form method="post" action="update" name="blog" object="{blog}">
        <label for="name">Title <span class="required">(required)</span></label><br />
        <f:form.textbox property="title" id="title" />
        <br />
        <label for="description">Description</label><br />
        <f:form.textarea property="description" rows="2" cols="40" id="description" />
        <br />
        <f:form.submit value="Update" />
    </f:form>
</f:section>

```

Most of this should already look familiar. However, there is a tiny difference to the `new` form you created earlier: in this edit form you added `object="{blog}"` to the `<f:form>` tag. This attribute binds the variable `{blog}` to the form and it simplifies the further definition of the form's elements. Each element - in our case the text box and the text area - comes with a `property` attribute declaring the name of the property which is supposed to be displayed and edited by the respective element.

Because you specified `property="title"` for the text box, Fluid will fetch the value of the blog's `title` property and display it as the default value for the rendered text box. The resulting `input` tag will also contain the name `"title"` due to the `property` attribute you defined. The `id` attribute only serves as a target for the `label` tag and is not required by Fluid.

Enough theory, let's try out the edit form in practice. A click on the `Edit` link of your list of blogs should result in a screen similar to this:



Edit blog "robert"

Update the information below:

Title (required)

Description



Before you can submit the form you need to implement the `updateAction`:

```
/**
 * Updates an existing blog
 *
 * @param \F3\Blog\Domain\Model\Blog $blog A not yet persisted clone of the
 * original blog containing the modifications
```

```

* @return void
*/
public function updateAction(\F3\Blog\Domain\Model\Blog $blog) {
    $this->blogRepository->update($blog);
    $this->pushFlashMessage('Your blog has been updated.');
```

Quite easy as well, isn't it? The `updateAction` expects the edited blog as its argument and passes it to the repository's `update` method. Before we disclose the secret how this magic actually works behind the scenes try out if updating the blog description really works:

Getting Started - Blog Example

our blog has been updated.

ere is a list of blogs:

Robert Lemke

Robert Lemke Code Artisanium [Edit](#) [Delete](#)

4.3 A Closer Look on Updates

Although updating objects is very simple on the user's side (that's where you live), it is a bit complex on behalf of the framework. You may skip this section if you like - but if you dare to take a quick look behind the scenes to get a better understanding of the mechanism behind the `updateAction` read on ...

The `updateAction` expects one argument, namely the *edited blog*. "Edited blog" means that this is a `Blog` object which already contains the values submitted by the edit form but is *not yet connected* to the repository in any way. At the time the `updateAction` receives the blog object two blogs with the same identity (i.e. with the same internal unique identifier) exist: One is the original, unmodified blog residing in the repository and the other one is a *clone* of the original blog with the new values already applied.

Cloning an entity object, such as a blog, with PHP's `clone` keyword creates an exact copy of the original with the only difference that the copy is not connected to the repository and therefore modifications to this instance will *not be persisted*. Consider the following example:

```

$blogA = $blogRepository->findByTitle('Robert Lemke');
$blogB = clone $blogA;

$blogA->setDescription('Modified');
$blogB->setDescription('Modified');
```

The new description of `$blogA` will be persisted automatically at the end of the request, all modification to `$blogB` however will be lost because it is only a clone.

Now that you know that the `blog` passed to the `updateAction` is a clone and therefore not stored in a repository, you might wonder how to replace the original blog object with the edited blog clone. The repository's `update` method does exactly that: it takes a clone, determines its technical identity, tries to find an object in the repository having the same identity and finally replaces the original by the clone.

The following two solutions are equivalent:

```
// using update():
$blogRepository->update($editedBlog);

// using replace():
$uuid = $persistenceBackend->getIdentifierByObject($editedBlog);
$originalBlog = $persistenceBackend->getObjectByIdentifier($uuid);
$blogRepository->replace($originalBlog, $editedBlog);
```

In some situations it is completely okay and even necessary to use the repository's `replace` method, for example if you want to replace an existing object by a completely new (i.e. not cloned) instance. However, if you know that you're dealing with a clone, always prefer `update`.

If all these details didn't scare you, you might now ask yourself how FLOW3 could know that the `updateAction` expects a clone and not the original? Great question. And the answer is - literally - hidden in the form generated by Fluid's form view helper:

```
<form method="post" name="blog" action="blog/blog/update">
  <input type="hidden" name="blog[__identity]"
    value="2d064493-ce45-4bc9-9d0c-38e40f2c4afe" />
  ...
</form>
```

Fluid automatically rendered a hidden field containing information about the technical identity of the form's object. This information is added in two cases:

- if the object is an original, previously retrieved from a repository
- if the object is a clone of an original

On receiving a request, the MVC framework checks if a special identity field (such as the above hidden field) is present and if further properties have been submitted. This results in three different cases:

▼ *Create, Show, Update detection*

Situation	Case	Consequence
identity missing, properties present	New / Create	Create a completely new object and set the given properties
identity present, properties missing	Show / Delete / ...	Retrieve original object with given identifier
identity present, properties present	Edit / Update	Retrieve original object, clone it and set the given properties

Because the edit form contained both identity and properties, FLOW3 prepared a clone with the given properties for our `updateAction`.

▼ Chapter 10: Validation

Hopefully the examples of the previous chapters made you shudder or at least raised some questions. Although it's surely nice to have one-liners for actions like `create` and `update` we need some more code to validate the incoming values before they are eventually persisted. You need to make sure that a blog title only consists of

regular characters and spaces, at least 3 and at maximum 50 (depending on your preference) and doesn't contain any HTML or other evil markup.

But do you really want all these checks in your action methods? Shouldn't we rather separate the concerns

Footnote See also: [Separation of Concerns \(Wikipedia\)](#)

of the action methods (show, create, update, ...) from others like validation, logging and security?

Fortunately FLOW3's validation framework doesn't ask you to add any additional PHP code to your action methods. Validation has been extracted as a separated concern which does it's job almost transparently to the developer.

▼ 1 Declaring Validation Rules

When we're talking about validation, we usually refer to validating *models*. The rules defining how a model should be validated can be classified into three types:

- **Base Properties** - a set of rules defining the minimum requirements on the properties of a model which must be met before a model may be persisted.
- **Base Model** - a set of rules or custom validator enforcing the minimum requirements on the combination of properties of a model which must be met before a model may be persisted.
- **Supplemental** - a set of rules defining additional requirements on a model for a specific situation, for example for a certain action method.

Note Base model and supplemental rules are not covered by this tutorial.

Rules for the base properties are defined directly in the model in form of annotations:

```
class Blog {

    /**
     * The blog's identifier.
     *
     * @var string
     * @validate Alphanumeric, StringLength(minimum = 3, maximum = 50)
     * @identity
     */
    protected $identifier = '';

    /**
     * The blog's title.
     *
     * @var string
     * @validate Text, StringLength(minimum = 1, maximum = 80)
     */
    protected $title = '';

    /**
     * A short description of the blog
     *
     * @var string
     * @validate Text, StringLength(maximum = 150)
```

```

 */
protected $description = '';

/**
 * The posts contained in this blog
 *
 * @var \SplObjectStorage<\F3\Blog\Domain\Model\Post>
 */
protected $posts;

...

```

The `@validate` annotations define one or more validation rules which should apply to a property. Rules are either separated by a comma or can be defined in dedicated lines by further `@validate` annotations.

Tip FLOW3 provides a range of built-in validators which can be found in the `FLOW3\Validation\Validator` sub package. The names used in the `@validate` declarations are just the class names of these validators. It is possible and very simple to program custom validators by implementing the `F3\FLOW3\Validation\Validator\ValidatorInterface`. Such validators must, however, be referred to by their fully qualified class name (i.e. including the namespace).

Please apply the above validation rules to your `Blog` model, click on the *Create another blog* link of your list of blogs and submit the empty form. If all went fine, you should end up again in the *new blog* form, with the tiny difference that the text boxes for the identifier and description are now framed in red:

Create a new blog

Enter information about your new blog below:

Identifier (required)

Title (required)

Description

2 Displaying Validation Errors

The validation rules seem to be in effect but the output could be a bit more meaningful. Please open the `new.html` template file again because we'd like to display a list of error messages for exactly this case when the form has been submitted but contained errors.

Fluid comes with a specialized view helper which allows for iterating over validation errors. Just add the `<f:form.errors>` view helper to your `new.html` template as shown in this example:

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

```

<f:section name="mainbox">
  <h2 class="flow3-firstHeader">Create a new blog</h2>
  <f:form.errors for="newBlog">
    <div class="error">
      <strong>{error.propertyName}</strong>:
      <f:for each="{error.errors}" as="errorDetail">{errorDetail.message}</f:for>
    </div>
  </f:form.errors>
  <p>Enter information about your new blog below:</p>

```

Similar to the `<f:for>` view helper `<f:form.errors>` defines a loop iterating over validation errors. The attribute `as` is optional and if it's not specified (like in the above example) `as="error"` is assumed.

To clearly understand this addition to the template you need to know that errors can be nested: There is a global error object containing the errors of the different domain objects (such as `newBlog`) which contain errors for each property which in turn can be multiple errors per property.

After saving the modified template and submitting the empty form again you should see some more verbose error messages:

reate a new blog

identifier: The length of the given string was not between 3 and 50 characters.

title: The length of the given string was not between 1 and 80 characters.

Enter information about your new blog below:

3 Validating Updated Arguments

Now that you know how validation errors can be displayed, you should add a `<f:form.errors>` view helper to the `edit.html` template as well:

```

<f:layout name="master" />

<f:section name="mainbox">
  <div class="flow3-header flow3-header-n1">
    <h1 class="flow3-firstHeader">Edit blog "{blog.identifier}"</h1>

```

```

</div>
<f:form.errors for="blog">
  <div class="error">
    <strong>{error.propertyName}</strong>:
    <f:for each="{error.errors}" as="errorDetail">{errorDetail.message}</f:for>
  </div>
</f:form.errors>
<p class="bodytext">Update the information below:</p>
...

```

Try updating a blog with an empty title and you should see the following:

```

W3 Exception
: Could not ultimately dispatch the request after 101 iterations. (More information)
C:\Exception\InfiniteLoop thrown in file
he2/htdocs/dev/flow3/dist/Packages/Framework/FLOW3/Classes/MVC/Dispatcher.php in line 93.
\MVC\Dispatcher::dispatch(P3\FLOW3\MVC\Web\Request, P3\FLOW3\MVC\Web\Response)

```

Can you imagine what happened? Let's look at the `editAction` again:

```

/**
 * Edits an existing blog
 *
 * @param \F3\Blog\Domain\Model\Blog $blog The blog to be edited.
 * @return string Form for editing the existing blog
 */
public function editAction(\F3\Blog\Domain\Model\Blog $blog) {
    $this->view->assign('blog', $blog);
}

```

When you started to edit the blog, the `editAction` received the original `Blog` object as its argument. You assigned this object to the Fluid template which displayed its current property values. Now you submitted the form with an empty title resulting in a new request, this time with the `updateAction` as its target.

Before the `updateAction` could be called, FLOW3 analyzed the incoming request. Because it recognized one argument as a `Blog` object, it invoked the respective validation rules - which failed due to the empty title. In these cases FLOW3 forwards the request to the referring action which is, in this case, the `editAction`.

The `editAction` expects a (valid) blog as its argument but unfortunately the blog is not valid. Because for FLOW3 this action call is like any other action call it does not execute the `editAction` but instead tries to dispatch the request to another action which can handle the error. This is, unfortunately, still the `editAction` which in the end results in an infinite loop.

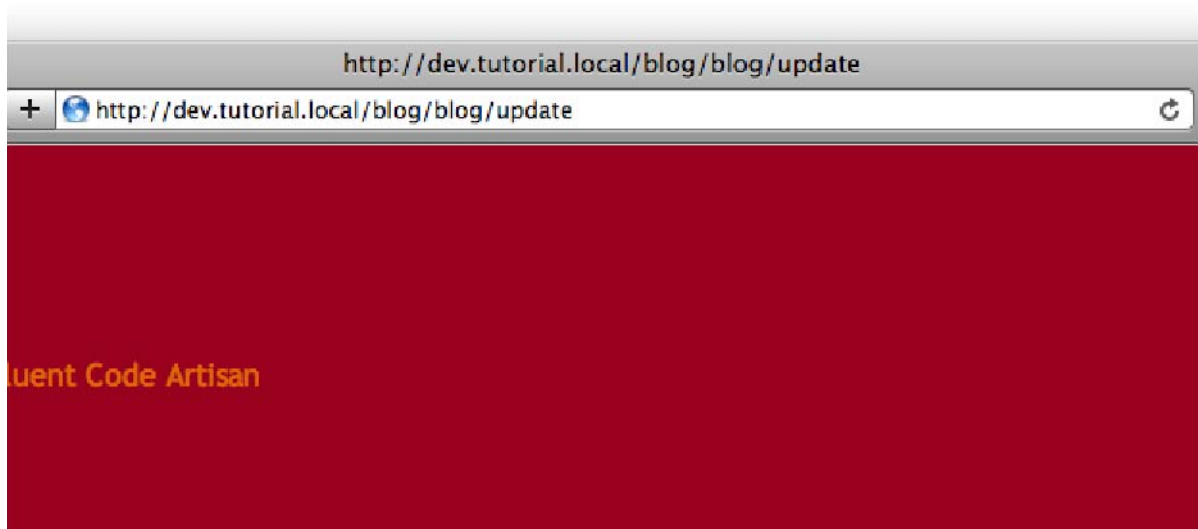
So the problem is that FLOW3 tries to validate the `$blog` argument for the `editAction` although we don't need a valid blog at this point. What's important is that the blog to `updateAction` or `createAction` is valid, but we don't really care about the `editAction` or `newAction` which only displays the form.

There's a very simple remedy to this problem: don't validate the blog. With one additional annotation the whole

mechanism works as expected:

```
/**
 * Edits an existing blog
 *
 * @param \F3\Blog\Domain\Model\Blog $blog The blog to be edited.
 * @return string Form for editing the existing blog
 * @dontvalidate $blog
 */
public function editAction(\F3\Blog\Domain\Model\Blog $blog) {
    $this->view->assign('blog', $blog);
}
```

Now the edit action can be called even though `$blog` is not valid and the error message is displayed above the edit form:



Edit blog "robert"

title: The length of the given string was not between 1 and 80 characters.

Update the information below:

title (required)

description

luent Code Artisan

Update

Powered by

Chapter 11: Routing

Although the basic functions like creating or updating a blog work well already, the URIs still have a little blemish.

The index of blogs can only be reached by the cumbersome address `http://dev.tutorial.local/blog/blog` and the URL for editing a blog refers to the blog's UUID instead of the human-readable identifier.

FLOW3's routing mechanism allows for beautifying these URIs by simple but powerful configuration options.

▼ 1 Blog Index Route

Our first task is to simplify accessing the list of blogs. For that you need to edit a file called `Routes.yaml` in the global `Configurations/` directory (located at the same level like the `Data` and `Packages` directories). This file already contains a few routes which we ignore for the time being.

Please insert the following configuration at the top of the file (before the `TYPO3CR` route) and make sure that you use spaces exactly like in the example (remember, spaces have a meaning in YAML files and tabs are not allowed):

```
--
name: 'Blogs'
uriPattern: '(blogs.html) '
defaults:
  @package:   Blog
  @controller: Blog
  @action:    index
  @format:    html
```

This configuration adds a new route "Blogs" to the list of routes (-- creates a new list item). The `uriPattern` property defines the pattern URIs must match for this route to become active. In this example empty URIs (i.e. `http://dev.tutorial.local/`) or an URI `http://dev.tutorial.local/blogs.html` would match – the round brackets make the `blogs.html` string optional.

If the URI matches, the route's default values for package, controller action and format are set and the request dispatcher will choose the right controller accordingly.

Try calling `http://dev.tutorial.local/` and `http://dev.tutorial.local/blogs.html` now – you should in both cases see the list of blogs produced by the `BlogController`'s `indexAction`.

▼ 2 Composite Routes

As you can imagine, you rarely define only one route per package and storing all routes in one file can easily become confusing. To keep the global `Routes.yaml` clean you may define sub routes which include - if their own URI pattern matches - further routes provided by your package.

The `TYPO3CR` sub route configuration for example includes further routes if the URI path starts with the string 'TYPO3CR'. Only the URI part contained in the less-than and greater-than signs will be passed to the sub routes:

```
##
# TYPO3CR subroutes

--
name: 'TYPO3CR'
uriPattern: 'typo3cr<TYPO3CRSubroutes>'
defaults:
  @format: 'html'
subRoutes:
  TYPO3CRSubroutes:
    package: TYPO3CR
```

Let's define a similar configuration for the `Blog` package. Please replace the YAML code you just inserted (the blog index route) by the following sub route configuration:

```
##
# Blog subroutes

--
name: 'Blog'
uriPattern: '<BlogSubroutes>'
defaults:
  @format: 'html'
subRoutes:
  BlogSubroutes:
    package: Blog
```

For this to work you need to create a new `Routes.yaml` file in the `Configuration` folder of your `Blog` package (`Packages/Application/Blog/Configuration/Routes.yaml`) and paste the route you already created:

```
# #
# Routes configuration for the Blog package #
# #
# #

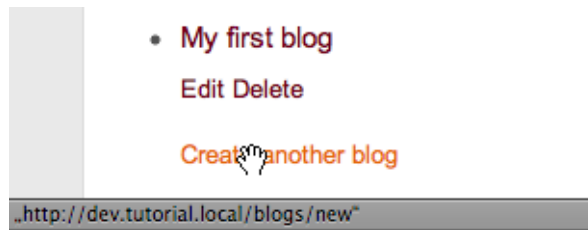
--
name: 'Blogs'
uriPattern: '(blogs.html) '
defaults:
  @package: Blog
  @controller: Blog
  @action: index
  @format: html
```

3 An Action Route

The URI pointing to the `newAction` is still `http://dev.tutorial.local/blog/blog/new` so let's beautify the action URIs as well by inserting a new route before the `'Blogs'` route:

```
--
name: 'Blog actions 1'
uriPattern: 'blogs/{@action}'
defaults:
  @package: Blog
  @controller: Blog
  @format: html
```

Reload the blog index and check out the new URI of the `createAction` - it's a bit shorter now:



However, the edit link still looks a bit ugly due to the blog's UUID

Footnote <http://dev.tutorial.local/blogs/edit?blog=e8262407-63b8-47a2-8b5f-2e3be9ecb077>

. For getting rid of this long identifier we need the help of a Route Part Handler.

4 Route Part Handlers

Route Part Handlers are classes which allow for custom conversion of arguments into URI parts and back. Our goal is to produce an URI like `http://dev.tutorial.local/blogs/robert/edit` and use this as our edit link.

Note At the time of this writing it is necessary to implement a custom route part handler for solving this task. However, we do plan to provide a generic route part handler which can be used at least for the simple cases like the one we're looking at now.

A route part handler must be able to

- convert a list (array) of arguments for a certain sub part into a URI part (resolve)
- convert a URI part back into a list (array) of arguments (match)

Please create a new folder `Blog/Classes/RoutePartHandlers/` and a new file called `BlogRoutePartHandler.php`. Then copy & paste the following code:

```
<?php
declare(ENCODING = 'utf-8');
namespace F3\Blog\RoutePartHandlers;

/**
 * Blog route part handler
 *
 * @scope prototype
 */
class BlogRoutePartHandler extends \F3\FLOW3\MVC\Web\Routing\DynamicRoutePart {

    /**
     * Resolves the identifier of the blog
     *
     * @param \F3\Blog\Domain\Model\Blog $value The Blog object
     * @return boolean TRUE if the name of the blog could be resolved and stored ↵
     *         in $this->value, otherwise FALSE.
     */
    protected function resolveValue($value) {
        if (!$value instanceof \F3\Blog\Domain\Model\Blog) return FALSE;
        $this->value = $value->getIdentifier();
        return TRUE;
    }
}
```

```

}

/**
 * While matching, converts the blog identifier into an identifier array
 *
 * @param string $value value to match, the blog identifier
 * @return boolean TRUE if value could be matched successfully, otherwise FALSE.
 */
protected function matchValue($value) {
    if ($value === NULL || $value === '') return FALSE;
    $this->value = array('__identity' => array('identifier' => $value));
    return TRUE;
}
}
?>

```

The method `resolveValue` will later receive a `Blog` object which its supposed to convert into a string suitable for being used in the URI. What this `resolveValue` implementation does is just use the blog's `identifier` property as the URI path segment. Of course it could equally have been the title or another property which looks nice in URIs.

The `matchValue` method on the other hand receives a part of the URI path which has been requested by the user. This part will be the blog's `identifier` property. For FLOW3 being able to recognize that the route part value needs to be converted into an object, a special `__identity` array needs to be created which in the end contains the `Blog` property "identifier".

Don't worry if you don't oversee this mechanism on the first glance, it really is an advanced topic. But we want beautified URIs from the beginning, don't we?

Now that you have created a custom route part handler we only need to include it into our routes configuration:

```

# #
# Routes configuration for the Blog package #
# #

--
name: 'Blog actions 2'
uriPattern: 'blogs/{blog}/{@action}'
defaults:
    @package:    Blog
    @controller: Blog
    @format:     html
routeParts:
    blog:
        handler: F3\Blog\RoutePartHandlers\BlogRoutePartHandler

--
name: 'Blog actions 1'
uriPattern: 'blogs/{@action}'

```

```

defaults:
  @package:   Blog
  @controller: Blog
  @format:    html

--

name: 'Blogs'
uriPattern: '(blogs.html)'
defaults:
  @package:   Blog
  @controller: Blog
  @action:    index
  @format:    html

```

The "Blog actions 2" route now handles all actions where a blog needs to be specified (i.e. show, edit, update and delete). In case the requested URI is `http://dev.tutorial.local/blogs/robert/edit`, the blog route part handler's method `matchValue` will be called with the parameter `robert` which then will be converted to the `Blog` object with just that identifier.

Finally, now that you copied and pasted so much code, you should try out the new routing setup:

Getting Started - Blog Example

Here is a list of blogs:

- **Robert Lemke**
Fluent Code Artisanium [Edit](#) [Delete](#)

`„http://dev.tutorial.local/blogs/robert/edit“`

5 More on Routing

The more an application grows, the more complex routing can become and sometimes you'll wonder which route FLOW3 eventually chose. One way to get this information is looking at the log file which is by default located in `Data/Logs/Web/`:

```

mc - /Users/Shared/Sites — tail — 145x8
il -f FLOW3_Development.log
FLOW3      --- Launching FLOW3 in Development context. ---
G FLOW3      Router route(): Route "Blog :: Blog actions 2" matched the request path "/blo
G FLOW3      Dispatching signal F3\FLOW3\Core\Bootstrap::emitFinishedNormalRun ...
G FLOW3      to slot F3\FLOW3\Core\LockManager::unlockSite.
FLOW3      Shutting down ...

```

More information on routing can be found in the [FLOW3 reference manual](#).

▼ Chapter 12: Summary

▼ 1 Next Steps

This is the end of the Getting Started Tutorial. You now got a first impression of how a FLOW3 application looks like and how the most important modules of FLOW3 work together.

You now have two options for delving more into FLOW3 programming:

1. Start completing the missing functionality on your own and while you do, read further parts of the FLOW3 reference manual
2. Install the finished blog example and explore its code by reading and modifying it

If you can't wait to see the finished blog example all you need to do is:

1. Delete everything within your blog package (that is everything below `Packages/Application/Blog/`)
2. Copy everything from `Packages/Application/GettingStarted/Resources/Private/CheatSheet/` to the Blog package directory

▼ 2 Feedback

This is the very first tutorial about FLOW3 and me and the FLOW3 core team are curious about getting your feedback! If you have any questions, are stuck at some point or just want to let me know how you liked the tutorial please write to the [FLOW3 mailing list](#) or drop me a line via robert@typo3.org.

And if you love FLOW3 like we do, spread the word in your blog or through your favorite social network ...