Flow Framework

Release 6.3.x

Neos Team and Contributors

Apr 05, 2023

CONTENTS

Flow is a free PHP framework licensed under the MIT license, developed to power the enterprise Neos CMS.

This version of the documentation covering Flow 6.3.x has been rendered at: Apr 05, 2023

Note: We'd love to get your feedback on this documentation! Please share your thoughts in our forum, or the #flow-general channel in the Neos Project's Slack.

Help is always greatly appreciated, read *Contributing to Flow* to find out how you can improve Flow.

CHAPTER

ONE

QUICKSTART

1.1 What Is in This Guide?

This guided tour gets you started with Flow by giving step-by-step instructions for the development of a small sample application. It will give you a first overview of the basic concepts and leaves the details to the full manual and more specific guides.

Be warned that your head will be buzzed with several new concepts. But after you made your way through the whitewater you'll surely ride the wave in no time!

1.2 What Is Flow?

Flow is a PHP-based application framework which is especially well-suited for enterprise-grade applications. Its architecture and conventions keep your head clear and let you focus on the essential parts of your application. Although stability, security and performance are all important elements of the framework's design, the fluent user experience is the one underlying theme which rules them all.

As a matter of fact, Flow is easier to learn for PHP beginners than for veterans. It takes a while to leave behind old paradigms and open up for new approaches. That being said, developing with Flow is very intuitive and the basic principles can be learned within a few hours. Even if you don't decide to use Flow for your next project, there are a lot of universal development techniques you can learn.

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

1.3 Installing Flow

Setting up Flow is pretty straight-forward. As a minimum requirement you will need:

- A web server (we recommend Apache with the *mod_rewrite* module enabled)
- PHP 7.2.0 or later
- A database supported by Doctrine DBAL, such as MySQL
- Command line access

Install Composer by following the installation instructions which boils down to this in the simplest case:

```
curl -s https://getcomposer.org/installer | php
```

Note: Feel free to install the composer command to a global location, by moving the phar archive to e.g. */usr/local/bin/composer* and making it executable. The following documentation assumes `composer` is installed globally.

Tip: Running `composer selfupdate` from time to time keeps it up to date and can prevent errors caused by composer not understanding e.g. new syntax in manifest files.

Then use Composer in a directory which will be accessible by your web server to download and install all packages of the Flow Base Distribution. The following command will clone the latest version, include development dependencies and keep git metadata for future use:

```
composer create-project --keep-vcs neos/flow-base-distribution Quickstart
```

You will end up with a directory structure like this:

```
htdocs/               <-- depending on your web server
  Quickstart/
    Build/
    Configuration/
      Settings.yaml.example
      ...
    Packages/
      Framework/
        Neos.Flow/
        ...
    Web/              <-- your virtual host root will point to this
      .htaccess
      index.php
    flow
    flow.bat
```

1.4 Setting File Permissions

You will access Flow from both, the command line and the web browser. In order to provide write access to certain directories for both, you will need to set the file permissions accordingly. But don't worry, this is simply done by changing to the Flow base directory (`Quickstart` in the above example) and calling the following command:

command line:

```
./flow core:setfilepermissions john www-data www-data
```

Please replace *john* by your own username. The second argument is supposed to be the username of your web server and the last one specifies the web server's group. For most installations on Mac OS X this would be both *_www* instead of *www-data*.

It can and usually will happen that Flow is launched from the command line by a different user. All users who plan using Flow from the command line need to join the web server's group. On a Linux machine this can be done by typing:

command line:

```
sudo usermod -a -G www-data john
```

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

command line:

```
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: Setting file permissions is not necessary and not possible on Windows machines. For Apache to be able to create symlinks, you need to use Windows Vista (or newer) and Apache needs to be started with Administrator privileges.

1.5 Setting up a virtual host

It is very much recommended to create a virtual host configuration for Apache that uses the *Web* folder as the document root. This has a number of reasons:

- it makes for nicer URLs
- it is **more secure** because that way access to anything else through the web is not possible

The latter point is really important!

For the rest of this tutorial we assume you have created a virtual host that can be reached through `http://quickstart/`.

1.6 Testing the Installation

If your system is configured correctly you should now be able to access the Welcome screen:

```
http://quickstart/
```

If you did not follow our advice to create a virtual host, point your browser to the `Web` directory of your Flow installation throughout this tutorial, for example:

```
http://localhost/Quickstart/Web/
```

The result should look similar to the screen you see in the screenshot. If something went wrong, it usually can be blamed on a misconfigured web server or insufficient file permissions.

Note: If all you get is a 404, you might need to edit the `.htaccess` file in the `Web` folder to adjust the `RewriteBase` directive as needed.

Note: Depending on your environment (especially on Windows systems) you might need to set the path to the PHP binary in `Configuration/Settings.yaml`. If you copied the provided example Settings you only need to uncomment the corresponding lines and adjust the path.

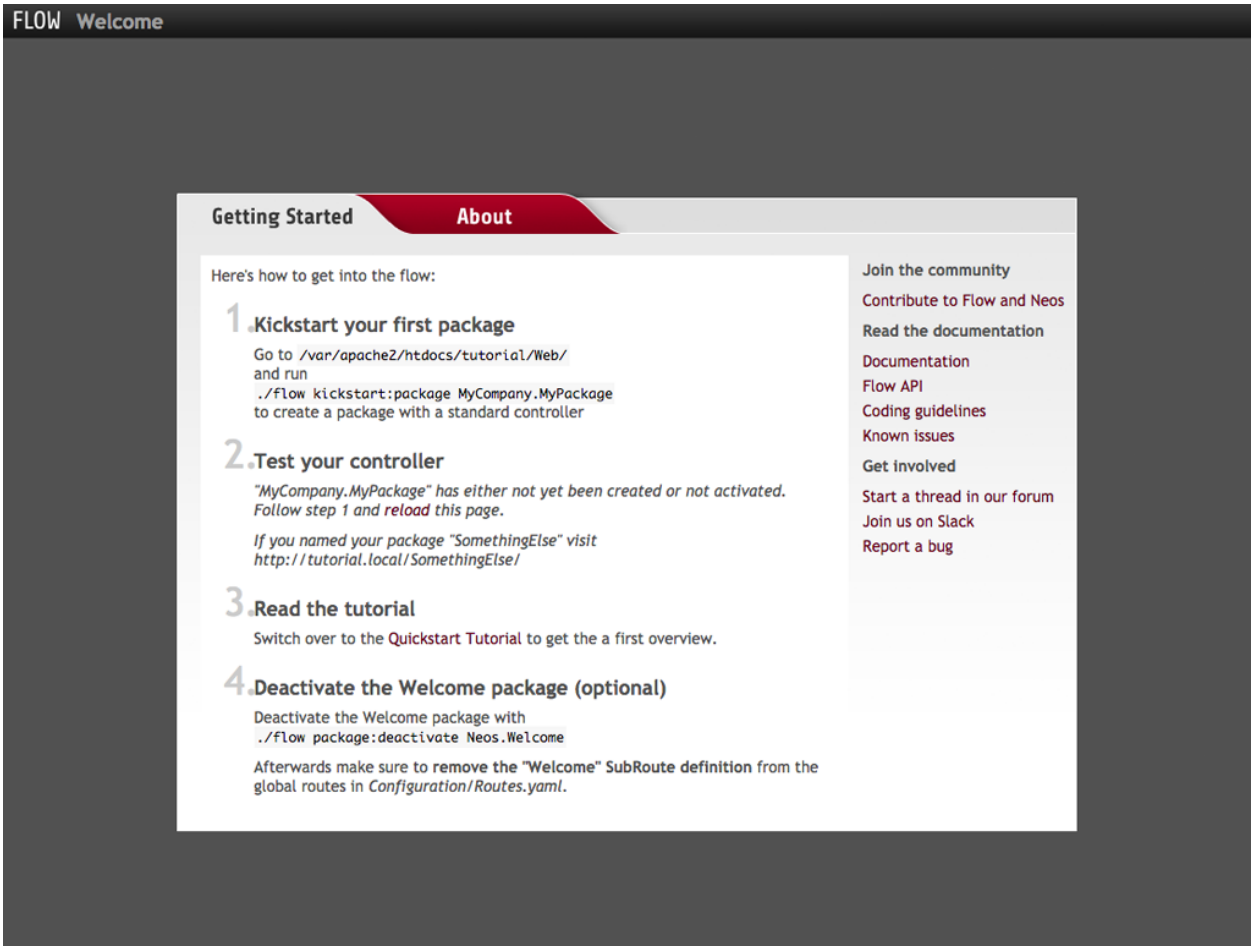

Fig. 1: The Flow Welcome Screen

Tip: There are some friendly ghosts in our Slack channel and in the Discuss forum – they will gladly help you out if you describe your problem as precisely as possible.

Some Note About Speed

The first request will usually take quite a while because Flow does a lot of heavy lifting in the background. It analyzes code, builds up reflection caches and applies security rules. During all the following examples you will work in the so called *Development Context*. It makes development very convenient but feels a lot slower than the *Production Context* – the one you will obviously use for the application in production.

1.7 Kickstarting a Package

The actual code of an application and its resources – such as images, style sheets and templates – are bundled into *packages*. Each package is identified by a globally unique package key, which consists of your company or domain name (the so called *vendor name*) and further parts you choose for naming the package.

Let's create a *Demo* package for our fictive company *Acme*:

```
$ ./flow kickstart:package Acme.Demo
Created .../Acme.Demo/Classes/Acme/Demo/Controller/StandardController.php
Created .../Acme.Demo/Resources/Private/Layouts/Default.html
Created .../Acme.Demo/Resources/Private/Templates/Standard/Index.html
```

The Kickstarter will create a new package directory in *Packages/Application/* resulting in the following structure:

```
Packages/
  Application/
    Acme.Demo/
      Classes/Acme/Demo/
      Configuration/
      Documentation/
      Meta/
      Resources/
      Tests/
```

The **`kickstart:package`** command also generates a sample controller which displays some content. You should be able to access it through the following URL:

```
http://quickstart/Acme.Demo
```

Tip: In case your web server lacks mod_rewrite, it could be that you need to call this to access the controller:

```
http://quickstart/index.php/Acme.Demo
```

If this the case, keep in mind to add `index.php` to the following URLs in this Quickstart tutorial.

1.8 Hello World

Let's use the *StandardController* for some more experiments. After opening the respective class file in *Packages/Application/Acme.Demo/Classes/Acme/Demo/Controller/* you should find the method *indexAction()* which is responsible for the output you've just seen in your web browser:

```
/**
 * @return void
 */
public function indexAction() {
    $this->view->assign('foos', array(
        'bar', 'baz'
    ));
}
```

Accepting some kind of user input is essential for most applications and Flow does a great deal of processing and sanitizing any incoming data. Try it out – create a new action method like this one:

```
/**
 * This action outputs a custom greeting
 *
 * @param string $name your name
 * @return string custom greeting
 */
public function helloAction($name) {
    return 'Hello ' . $name . '!';
}
```

Important: For the sake of simplicity the above example does not contain any input/output sanitation. If your controller action directly returns something, make sure to filter the data!

Tip: You should always properly document all your functions and class properties. This will not only help other developers to understand your code, but is also essential for Flow to work properly.

Now test the new action by passing it a name like in the following URL:

```
http://quickstart/Acme.Demo/Standard/hello?name=Robert
```

The path segments of this URL tell Flow to which controller and action the web request should be dispatched to. In our example the parts are:

- *Acme.Demo* (package key)
- *Standard* (controller name)
- *hello* (action name)

If everything went fine, you should be greeted by a friendly "*Hello Robert!*" – if that's the name you passed to the action. Also try leaving out the *name* parameter in the URL – Flow will complain about a missing argument.

1.9 Database Setup

One important design goal for Flow was to let a developer focus on the business logic and work in a truly object-oriented fashion. While you develop a Flow application, you will hardly note that content is actually stored in a database. Your code won't contain any SQL query and you don't have to deal with setting up table structures.

But before you can store anything, you still need to set up a database and tell Flow how to access it. The credentials and driver options need to be specified in the global Flow settings.

After you have created an empty database and set up a user with sufficient access rights, copy the file *Configuration/Settings.yaml.example* to *Configuration/Settings.yaml*. Open and adjust the file to your needs – for a common MySQL setup, it would look similar to this:

```
Neos:
  Flow:
    persistence:
      backendOptions:
        driver: 'pdo_mysql'
        dbname: 'quickstart' # adjust to your database name
        user: 'root'         # adjust to your database user
        password: 'password' # adjust to your database password
        host: '127.0.0.1'    # adjust to your database host
```

Note: If you are not familiar with the *YAML* format yet, there are two things you should know at least:

- Indentation has a meaning: by different levels of indentation, a structure is defined.
- Spaces, not tabs: you must indent with exactly 2 spaces per level, don't use tabs.

If you configured everything correctly, the following command will create the initial table structure needed by Flow:

```
$ ./flow doctrine:migrate
Migrating up to 2011xxxxx00 from 0

++ migrating 2011xxxxx00
    -> CREATE TABLE flow_resource_resourcepointer (hash VARCHAR(255) NOT NULL, PRIMARY
    -> CREATE TABLE flow_resource_resource (persistence_object_identifier VARCHAR(40)
...
++ finished in 0.76
```

1.10 Storing Objects

Let's take a shortcut here – instead of programming your own controller, model and view just generate some example with the kickstarter:

```
$ ./flow kickstart:actioncontroller --generate-actions --generate-related Acme.Demo␣
↪CoffeeBean
Created .../Acme.Demo/Classes/Acme/Demo/Domain/Model/CoffeeBean.php
Created .../Acme.Demo/Tests/Unit/Domain/Model/CoffeeBeanTest.php
Created .../Acme.Demo/Classes/Acme/Demo/Domain/Repository/CoffeeBeanRepository.php
Created .../Acme.Demo/Classes/Acme/Demo/Controller/CoffeeBeanController.php
Omitted .../Acme.Demo/Resources/Private/Layouts/Default.html
Created .../Acme.Demo/Resources/Private/Templates/CoffeeBean/Index.html
Created .../Acme.Demo/Resources/Private/Templates/CoffeeBean/New.html
```

(continues on next page)

(continued from previous page)

```
Created .../Acme.Demo/Resources/Private/Templates/CoffeeBean/Edit.html
Created .../Acme.Demo/Resources/Private/Templates/CoffeeBean/Show.html
As new models were generated, do not forget to update the database schema with the␣
↪respective doctrine:* commands.
```

Whenever a model is created or modified, the database structure needs to be adjusted to fit the new PHP code. This is something you should do consciously because existing data could be altered or removed – therefore this step isn't taken automatically by Flow.

The kickstarter created a new model representing a coffee bean. For promoting the new structure to the database, just run the **`doctrine:update`** command:

```
$ ./flow doctrine:update
Executed a database schema update.
```

Tip: In a real project you should avoid the **`doctrine:update`** command and instead work with migrations. See the "Persistence" section of the *The Definitive Guide* for more details

---

A quick glance at the table structure (using your preferred database management tool) will reveal that a new table for coffee beans has been created.

The controller rendered by the kickstarter provides some very basic functionality for creating, editing and deleting coffee beans. Try it out by accessing this URL:

```
http://quickstart/Acme.Demo/CoffeeBean
```

Create a few coffee beans, edit and delete them and take a look at the database tables if you can't resist ...

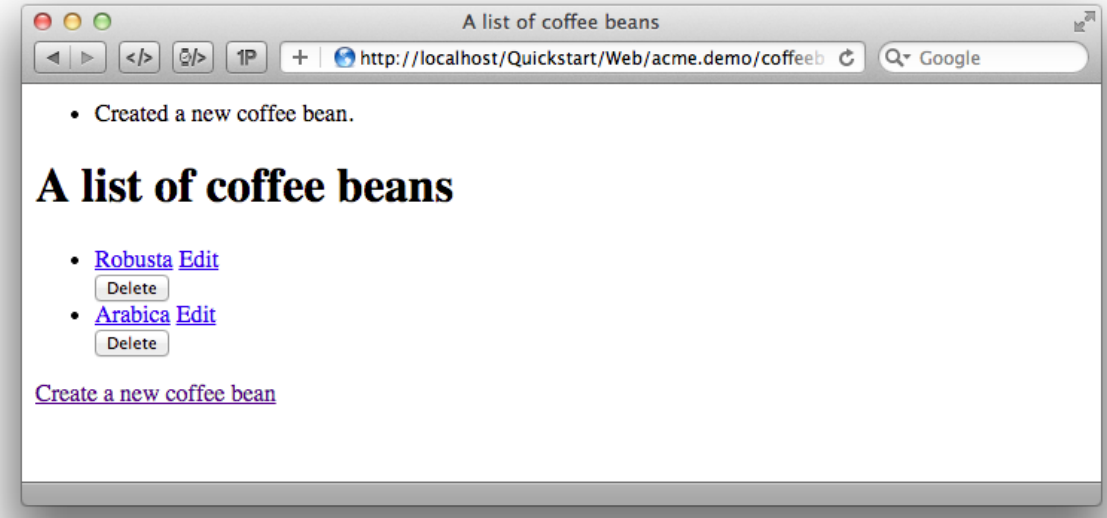

Fig. 2: List and create coffee beans

1.11 A Closer Look at the Example

In case you have been programming PHP for a while, you might be used to tackle many low-level tasks yourself: Rendering HTML forms, retrieving and validating input from the superglobals `$_GET`, `$_POST` and `$_FILES`, validating the input, creating SQL queries for storing the input in the database, checking for Cross-Site Scripting, Cross-Site Request Forgery, SQL-Injection and much more.

With this background, the following complete code listing powering the previous example may seem a bit odd, if not magical to you. Take a close look at each of the methods – can you imagine what they do?

```
use Acme\Demo\Domain\Model\CoffeeBean;
use Acme\Demo\Domain\Repository\CoffeeBeanRepository;

class CoffeeBeanController extends ActionController {

    /**
     * @Flow\Inject
     * @var CoffeeBeanRepository
     */
    protected $coffeeBeanRepository;

    /**
     * @return void
     */
    public function indexAction() {
        $this->view->assign('coffeeBeans', $this->coffeeBeanRepository->findAll());
    }

    /**
     * @param CoffeeBean $coffeeBean
     * @return void
     */
    public function showAction(CoffeeBean $coffeeBean) {
        $this->view->assign('coffeeBean', $coffeeBean);
    }

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

    /**
     * @param CoffeeBean $newCoffeeBean
     * @return void
     */
    public function createAction(CoffeeBean $newCoffeeBean) {
        $this->coffeeBeanRepository->add($newCoffeeBean);
        $this->addFlashMessage('Created a new coffee bean.');
        $this->redirect('index');
    }

    /**
     * @param CoffeeBean $coffeeBean
     * @return void
     */
    public function editAction(CoffeeBean $coffeeBean) {
```

(continues on next page)

(continued from previous page)

```
        $this->view->assign('coffeeBean', $coffeeBean);
    }

    /**
     * @param CoffeeBean $coffeeBean
     * @return void
     */
    public function updateAction(CoffeeBean $coffeeBean) {
        $this->coffeeBeanRepository->update($coffeeBean);
        $this->addFlashMessage('Updated the coffee bean.');
        $this->redirect('index');
    }

    /**
     * @param CoffeeBean $coffeeBean
     * @return void
     */
    public function deleteAction(CoffeeBean $coffeeBean) {
        $this->coffeeBeanRepository->remove($coffeeBean);
        $this->addFlashMessage('Deleted a coffee bean.');
        $this->redirect('index');
    }

}
```

You will learn all the nitty-gritty details of persistence (that is storing and retrieving objects in a database), Model-View Controller and validation in *The Definitive Guide*. With some hints for each of the actions of this controller though, you'll get some first impression of how basic operations like creating or deleting objects are handled in Flow.

Without further ado let's take a closer look at some of the actions:

1.11.1 indexAction

The `indexAction` displays a list of coffee beans. All it does is fetching all existing coffee beans from a *repository* and then handing them over to the template for rendering.

The `CoffeeBeanRepository` takes care of storing and finding stored coffee beans. The simplest operation it provides is the `findAll()` method which returns a list of all existing `CoffeeBean` objects.

For consistency reasons only one instance of the `CoffeeBeanRepository` class may exist at a time. Otherwise there would be multiple repositories storing `CoffeeBean` objects – and which one would you then ask for retrieving a specific coffee bean back from the database? The `CoffeeBeanRepository` is therefore tagged with an *annotation* stating that only a single instance may exist at a time:

```
/**
 * @Flow\Scope("singleton")
 */
class CoffeeBeanRepository extends Repository {
```

Because PHP doesn't support the concept of annotations natively, we are using doc comments which are parsed by an annotation parser in Flow.

Flow's object management detects the `Scope` annotation and takes care of all the details. All you need to do in order to get the right `CoffeeBeanRepository` instance is telling Flow to *inject* it into a class property you defined:

```
/**
 * @Flow\Inject
 * @var CoffeeBeanRepository
 */
protected $coffeeBeanRepository;
```

The `Inject` annotation tells Flow to set the `$coffeeBeanRepository` right after the `CoffeeBeanController` class has been instantiated.

Tip: This feature is called *Dependency Injection* and is an important feature of Flow. Although it is blindingly easy to use, you'll want to read some more about it later in the *related section* of the main manual.

Flow adheres to the Model-View-Controller pattern – that's why the actual output is not generated by the action method itself. This task is delegated to the *view*, and that is, by default, a *Fluid* template (Fluid is the name of the templating engine Flow uses). Following the conventions, there should be a directory structure in the `Resources/Private/Templates/` folder of a package which corresponds to the controllers and actions. For the `index` action of the `CoffeeBeanController` the template `Resources/Private/Templates/CoffeeBean/Index.html` will be used for rendering.

Templates can display content which has been assigned to *template variables*. The placeholder `{name}` will be replaced by the actual value of the template variable `name` once the template is rendered. Likewise `{coffeeBean.name}` is substituted by the value of the coffee bean's `name` attribute.

The coffee beans retrieved from the repository are assigned to the template variable `coffeeBeans`. The template in turn uses a for-each loop for rendering a list of coffee beans:

```
<ul>
    <f:for each="{coffeeBeans}" as="coffeeBean">
        <li>
            {coffeeBean.name}
        </li>
    </f:for>
</ul>
```

1.11.2 showAction

The `showAction` displays a single coffee bean:

```
/**
 * @param CoffeeBean $coffeeBean The coffee bean to show
 * @return void
 */
public function showAction(CoffeeBean $coffeeBean) {
    $this->view->assign('coffeeBean', $coffeeBean);
}
```

The corresponding template for this action is stored in this file:

```
Acme.Demo/Resources/Private/Templates/CoffeeBean/Show.html
```

This template produces a simple representation of the `coffeeBean` object. Similar to the `indexAction` the coffee bean object is assigned to a Fluid variable:

```
$this->view->assign('coffeeBean', $coffeeBean);
```

The `showAction` method requires a `CoffeeBean` object as its method argument. But we need to look into the template of the `indexAction` again to understand how coffee beans are actually passed to the `showAction`.

In the list of coffee beans, rendered by the `indexAction`, each entry links to the corresponding `showAction`. The links are rendered by a so-called *view helper* in the Fluid template `Index.html`:

```
<f:link.action action="show" arguments="{coffeeBean: coffeeBean}">...</f:link.action>
```

The interesting part is the `{coffeeBean: coffeeBean}` argument assignment: It makes sure that the `CoffeeBean` object, stored in the `coffeeBean` template variable, will be passed to the `showAction` through a GET parameter.

Of course you cannot just put a PHP object like the coffee bean into a URL. That's why the view helper will render an address like the following:

```
http://quickstart/acme.demo/coffeebean/show?
    coffeeBean%5B__identity%5D=910c2440-ea61-49a2-a68c-ee108a6ee429
```

Instead of the real PHP object, its *Universally Unique Identifier* (UUID) was included as a GET parameter.

Note: That certainly is not a beautiful URL for a coffee bean – but you'll learn how to create nice ones in the main manual.

Before the `showAction` method is actually called, Flow will analyze the GET and POST parameters of the incoming HTTP request and convert identifiers into real objects again. By its UUID the coffee bean is retrieved from the `CoffeeBeanRepository` and eventually passed to the action method:

```
public function showAction(CoffeeBean $coffeeBean) {
```

1.11.3 newAction

The `newAction` contains no PHP code – all it does is displaying the corresponding Fluid template which renders a form.

1.11.4 createAction

The `createAction` is called when a form displayed by the `newAction` is submitted. Like the `showAction` it expects a `CoffeeBean` as its argument:

```
/**
 * @param \Acme\Demo\Domain\Model\CoffeeBean $newCoffeeBean
 * @return void
 */
public function createAction(CoffeeBean $newCoffeeBean) {
    $this->coffeeBeanRepository->add($newCoffeeBean);
    $this->addFlashMessage('Created a new coffee bean.');
    $this->redirect('index');
}
```

This time the argument contains not an existing coffee bean but a new one. Flow knows that the expected type is `CoffeeBean` (by the type hint in the method and the param annotation) and thus tries to convert the POST data sent by the form into a new `CoffeeBean` object. All you need to do is adding it to the Coffee Bean Repository.

1.11.5 editAction

The purpose of the `editAction` is to render a form pretty much like that one shown by the `newAction`. But instead of empty fields, this form contains all the data from an existing coffee bean, including a hidden field with the coffee bean's UUID.

The edit template uses Fluid's form view helper for rendering the form. The important bit for the edit form is the form object assignment:

```
<f:form action="update" object="{coffeeBean}" objectName="coffeeBean">
    ...
</f:form>
```

The `object="{coffeeBean}"` attribute assignment tells the view helper to use the `coffeeBean` template variable as its subject. The individual form elements, such as the text box, can now refer to the coffee bean object properties:

```
<f:form.textfield property="name" id="name" />
```

On submitting the form, the user will be redirected to the `updateAction`.

1.11.6 updateAction

The `updateAction` receives the modified coffee bean through its `$coffeeBean` argument:

```
/**
 * @param \Acme\Demo\Domain\Model\CoffeeBean $coffeeBean
 * @return void
 */
public function updateAction(CoffeeBean $coffeeBean) {
    $this->coffeeBeanRepository->update($coffeeBean);
    $this->addFlashMessage('Updated the coffee bean.');
    $this->redirect('index');
}
```

Although this method looks quite similar to the `createAction`, there is an important difference you should be aware of: The parameter passed to the `updateAction` is an already existing (that is, already *persisted*) coffee bean object with the modifications submitted by the user already applied.

Any modifications to the `CoffeBean` object will be lost at the end of the request unless you tell Flow explicitly to apply the changes:

```
$this->coffeeBeanRepository->update($coffeeBean);
```

This allows for a very efficient dirty checking and is a safety measure - as it leaves control over the changes in your hands.

Speaking about safety measures: it's important to know that Flow supports the notion of "safe request methods". According to the HTTP 1.1 specification, GET and HEAD requests should not modify data on the sever side. Since we consider this a good principle, Flow will not persist any changes automatically if the request method is "safe". So ... don't use regular links for deleting your coffee beans - send a POST or DELETE request instead.

1.12 Next Steps

Congratulations! You already learned the most important concepts of Flow development.

Certainly this tutorial will have raised more questions than it answered. Some of these concepts – and many more you will learn – take some time to get used to. The best advice I can give you is to expect things to be rather simple and not look out for the complicated solution (you know, the *not to see the wood for the trees* thing . . .).

Next you should experiment a bit with Flow on your own. After you've collected even more questions, I suggest reading the *Getting Started Tutorial*.

At the time of this writing, The Definitive Guide is not yet complete and still contains a few rough parts. Also the Getting Started Tutorial needs some love and restructuring. Still, it already may be a valuable source for further information and I recommend reading it.

Get in touch with the growing Flow community and make sure to share your ideas about how we can improve Flow and its documentation:

- Slack channel
- Discuss forum

I am sure that, if you're a passionate developer, you will love Flow – because it was made with you, the developer, in mind.

Happy Flow Experience!

Robert on behalf of the Neos team

CHAPTER

TWO

THE DEFINITIVE GUIDE

2.1 Part I: Introduction and Fundamentals

2.1.1 Introduction

What is Flow?

Flow is a web application platform enabling developers to create excellent web solutions. It gives you fast results. It is a reliable foundation for complex applications. And it is backed by one of the biggest PHP communities.

The Epic Forward

The Definitive Guide is meant to be a technical resource for documentation of both Flow usage as well as the theories, patterns and practices to be used in effective Flow development. While the community and the authors of this guide will remain objective when presenting concepts, the information found herein may be strongly biased both positively and negatively for and/or against other known software development methods and practices. While the practices adopted in this guide are not the only ones possible, nor necessarily the right ones for all projects, they are the generally accepted "Best Practices" that surround the design decisions and direction that have been taken by Flow and its contributors to date.

The fanatical adoption of the processes, procedures and methodologies as outlined in the guide will enabled you to work faster, smarter and produce the best possible results when working within the Flow framework. Flow was created to complete a missing piece not available to the PHP developer community. Many of the comparable systems found in various other languages are based on proprietary technologies or based on languages that require additional layers or systems to build and run applications. A primary reason for this was that due to some initial shortcomings of earlier versions of PHP, it was not accepted as an "Enterprise" language as opposed to a .NET or Java.

With the emergence of PHP 5.3 and the feature set it has brought with it, a better ecosystem of PHP frameworks is now possible. Flow aims to implement a set of software design and development principles that have been proven to produce organized, highly extensible applications which can evolve over time with the demands and changes of their domain.

Parts of The Guide

Part I: Introduction and Fundamentals

In this section, you will get an overview of the underlying patterns and practices that are implemented into Flow at its core. After reading this section, you should have a concise and informed understanding of theories and methodologies that are involved in building a Flow application using "Best Practices".

Part II: Getting Started

In Getting Started, you will learn how to get a Flow application setup and ready to go. You will also be introduced to the basic building blocks for a Flow application and its packages.

Part III: Manual

As is the case with any manual, this section will focus on how to use the various pieces and mechanisms found within Flow. This will include descriptions of what each component does and example code of how to use or implement it into your application.

Part IV: Deployment and Administration

Learning to build an application based on Flow is one thing, but equally important is understanding how to deploy your application into the wild, and then how to maintain and support it once it's live. The guide has dedicated an entire section to ensuring you know the ins and outs of publishing and maintaining an application built on Flow.

Part V: Appendixes

Any framework is only as good as its ability to communicate clearly on the frameworks intent and design to its community. While a ubiquitous language around design patterns helps, the appendixes section aim to make getting to specific documentation and topic references more efficient. This section is much more effective when used after having read through the guide, acting as a quick reference for previously learned concepts.

2.1.2 Object-Oriented Programming

Object-oriented programming is a Programming Paradigm, applied in Flow and the Packages built on it. In this section we will give an overview of the basic concepts of Object Orientation.

Programs have a certain purpose, which is - generally speaking - to solve a problem. "Problem" does not necessarily mean error or defect but rather an actual task. This Problem usually has a concrete counterpart in real life.

A Program could for example take care of the task of booking a cruise in the Indian Ocean. If so we obviously have a problem (a programmer that has been working to much and finally decided to go on vacation) and a program, promising recuperation by booking a coach on one of the luxury liners for him and his wife.

Object Orientation assumes that a concrete problem is to be solved by a program, and a concrete problem is caused by real objects. Therefore focus is on the object. This can be abstract of course: it will not be something as concrete as a car or a ship all the time, but can also be a reservation, an account or a graphical symbol.

objects are "containers" for data and corresponding functionality. The data of an object is stored in its **Properties**. The functionality is provided by **Methods**, which can for example alter the properties of the object. In regard to the cruise liner we can say, that it has a certain amount of coaches, a length and width and a maximum speed. Further it has methods to start the motor (and hopefully to stop it again also), change the direction as well as to increase thrust, for you can reach your holiday destination a bit faster.

Why Object Orientation after all?

Surely some users will ask themselves why they should develop object oriented in the first place. Why not (just like until now) keep on developing procedural, thus stringing together functions? Because procedural programming has some severe disadvantages:

- Properties and methods belonging together with regard to content can not be united. This methodology, called **Encapsulation** in Object Orientation, is necessary, if only because of clear arrangement.
- It is rather difficult to re-use code
- All properties can be altered everywhere throughout the code. This leads to hard-to-find errors.
- Procedural code gets confusing easily. This is called Spaghetti code.

Furthermore Object Orientation mirrors the real world: Real objects exist, and they all have properties and (most of them) methods. This fact is now represented in programming.

In the following we'll talk about the object ship. We'll invoke this object, stock it with coaches, a motor and other useful stuff. Furthermore, there will be functions, moving the ship, thus turning the motor on and off. Later we'll even create a luxury liner based on the general ship and equip it with a golf simulator and satellite TV.

On the following pages, we'll try to be as graphic as possible (but still semantically correct) to familiarize you with object orientation. There is a specific reason: The more you can identify with the object and its methods, the more open you'll be for the theory behind Object Oriented Programming. Both is necessary for successful programming – even though you'll often not be able to imagine the objects you'll later work with as clearly as in our examples.

Classes and Objects

Let's now take a step back and imagine there'd be a blueprint for ships in general. We now focus not the ship but this blueprint. It is called **class**, in this case it is the class `Ship`. In PHP this is written as follows;

PHP Code:

```
<?php

class Ship {

...

}

?>
```

Note: In this piece of code we kept noting the necessary PHP tags at the beginning and end. We will spare them in the following examples to make the listings a bit shorter.

The key word `class` opens the class and inside the curly brackets properties and methods are written. we'll now add these properties and methods:

PHP Code:

```
class Ship {

        public $name;
        public $coaches;
        public $engineStatus;
```

(continues on next page)

(continued from previous page)

```
        public $speed;


        function startEngine() {}
        function stopEngine() {}
        function moveTo($location) {}

}
```

Our ship now has a name (`$name`), a number of coaches (`$coaches`) and a speed (`$speed`). In addition we built in a variable, containing the status of the engine (`$engineStatus`). A real ship, of course, has much more properties, all important somehow – for our abstraction these few will be sufficient though. We'll focus on why every property is marked with the key word `public` further down.

Note: For methods and properties we use a notation called **lowerCamelCase**: The first letter is lower case and all other parts are added without blank or underscore in upper case. This is a convention used in Flow.

We can also switch on the engine (`startEngine()`), travel with the ship to the desired destination (`moveTo($location)`) and switch off the engine again (`stopEngine()`). Note that all methods are empty, i.e. we have no content at all. We'll change this in the following examples, of course. The line containing method name and (if available) parameters is called method signature or method head. Everything contained by the method ist called method body accordingly.

Now we'll finally create an object from our class. The class `ship` will be the blueprint and `$fidelio` the concrete object.

PHP Code:

```
$fidelio = new Ship();

// Display the object
var_dump($fidelio);
```

The key word new is used to create a concrete object from the class. This object is also called **Instance **and the creation process consequentially **Instantiation**. We can use the command `var_dump()` to closely examine the object. We'll see the following

PHP Code:

```
object(Ship)#1 (3) {

        ["name"] => NULL

        ["coaches"] => NULL

        ["engineStatus"] => NULL

        ["speed"] => NULL

}
```

We can clearly see that our object has 4 properties with a concrete value, at the moment still NULL, for we did not yet assign anything. We can instantiate as many objects from a class as we like, and every single one will differ from the others – even if all of the properties have the same values.

PHP Code:

```
$fidelio1 = new Ship();
$fidelio2 = new Ship();

if ($fidelio1 === $fidelio2) {
        echo 'objects are identical!'
} else {
        echo 'objects are not identical!'
}
```

In this example the output is `objects are not identical!`

The arrow operator

We are able to create an object now, but of course it's properties are still empty.We'll hurry to change this by assigning values to the properties. For this, we use a special operator, the so called arrow operator (->). We can use it for getting access to the properties of an object or calling methods. In the following example, we set the name of the ship and call some methods:

PHP Code:

```
$ship = new Ship();
$ship->name = "FIDELIO";

echo "The ship's Name is ". $ship->name;

$ship->startEngine();
$ship->moveTo('Bahamas');
$ship->stopEngine();
```

$this

Using the arrow operator we can now comfortably access properties and methods of an object. But what to do, if we want to do this from inside a method, e.g. to set `$speed ``inside of the method ``startEngine()`? We don't know at this point, how an object to be instantiated later will be called. So we need a mechanism to do this independent from the name. This is done with the special variable `$this`.

PHP Code:

```
class Ship {

        ...

        public $speed;

        ...

        function startEngine() {

                $this->speed = 200;

        }

}
```

With `$this->speed` you can access the property *speed* in the actual object, independently of it's name.

Constructor

It can be very useful to initialize an object at the Moment of instantiating it. Surely there will be a certain number of coaches built in right away, when a new cruise liner is created - so that the future guest will not be forced to sleep in emergency accommodation. So we can define the number of coaches right when instantiating. The processing of the given value is done in a method automatically called on creation of an object, the so called **Constructor**. This special method always has the name `__construct()` (the first two characters are underscores).

The values received from instantiating are now passed on to the constructor as Argument and then assigned to the properties `$coaches ``respectively ``$name`.

Inheritance of Classes

With the class we created we can already do a lot. We can create many ships and send them to the oceans of the world. But of course the shipping company always works on improving the offer of cruise liners. Increasingly big and beautiful ships are built. Also new offers for the passengers are added. FIDELIO2, for example, even has a little golf course based on deck.

If we look behind the curtain of this new luxury liner though, we find that the shipping company only took a ship type FIDELIO and altered it a bit. The basis is the same. Therefore it makes no sense to completely redefine the new ship – instead we use the old definition and just add the golf course – just as the shipping company did. Technically speaking we extend an "old" class definition by using the key word `extends`.

PHP Code:

```
class LuxuryLiner extends Ship {

        public $luxuryCoaches;

        function golfSimulatorStart() {

                echo 'Golf simulator on ship ' . $this->name . '
                started.';

        }

        function golfSimulatorStop() {

                echo 'Golf simulator on ship ' . $this->name . '
                stopped.';

        }

}

$luxuryShip = new LuxuryLiner('FIDELIO2','600')
```

Our new luxury liner comes into existence as easy as that. We define, that the luxury liner just extends the Definition of the class `Ship`. The extended class (in or example `Ship`) is called **parent class **or **superclass**. The class formed by Extension (in our example `LuxuryLiner`) is called **child class **or **sub class**.

The class `LuxuryLiner` now contains the complete configuration of the base class `Ship` (including all properties and methods) and defines additional properties (like the amount of luxury coaches in `$luxuryCoaches`) and additional methods (like `golfSimulatorStart()` and `golfSimulatorStop()`). Inside these methods you can again access the properties and methods of the parent class by using `$this`.

Overriding Properties and Methods

Inside an inherited class you can not only access properties and methods of the parent class or define new ones. It's even possible to override the original properties and methods. This can be very useful, e.g. for giving a method of a child class a new functionality. Let's have a look at the method `startEngine()` for example:

PHP Code:

```
class Ship {
    ...
    $engineStatus = 'OFF';
    ...
    function startEngine() {
            $this->engineStatus = 'ON';
    }
    ...
}

class Luxusliner extends Ship {
    ...
    $additionalEngineStatus = 'OFF';
    ...
    function startEngine() {
            $this->engineStatus = 'ON';
            $this->additionalEngineStatus = 'ON';
    }
    ...
}
```

Our luxury liner (of course) has an additional motor, so this has to be switched on also, if the method `startEngine()` is called. The child class now overrides the method of the parent class and so only the method `startEngine()` of the child class is called.

Access to the parent class through "parent"

Overriding a method comes in handy, but has a serious disadvantage. When changing the method `startEngine()` in the parent class, we'd also have to change the method in the child class. This is not only a source for errors but also kind of inconvenient. It would be better to just call the method of the parent class and then add additional code before or after the call. That's exactly what can be done by using the key word `parent`. With `parent::methodname()` the method of the parent class can be accessed comfortably - so our former example can be re-written in a smarter way:

PHP Code:

```
class Ship {
    ...
    $engineStatus = 'OFF';
    ...
    function startEngine() {
            $this->engineStatus = 'ON';
    }
    ...
}

class Luxusliner extends Ship {
    ...
```

(continues on next page)

(continued from previous page)

```
    $additionalEngineStatus = 'OFF';
    ...
    function startEngine() {
            parent::startEngine();
            $this->additionalEngineStatus = 'ON';
    }
    ...
}
```

Abstract classes

Sometimes it is useful to define "placeholder methods" in the parent class which are filled in the child class. These "placeholders" are called **abstract methods**. A class containing abstract methods is called **abstract class**. For our ship there could be a method `setupCoaches()`. Each type of ship is to be handled differently for each has a proper configuration. So each ship must have such a method but the concrete implementation is to be done separately for each ship type.

PHP Code:

```
abstract class Ship {
...
    function __construct() {
            $this->setupCoaches();
    }
    abstract function setupCoaches();
...
}

class Luxusliner extends Ship {
...
    function setupCoaches() {
            echo 'Coaches are being set up';
    }
}

$luxusschiff = new Luxusliner();
```

In the parent class we have defined only the body of the method `setupCoaches()`. The key word `abstract` makes sure that the method must be implemented in the child class. So using abstract classes, we can define which methods have to be present later without having to implement them right away.

Interfaces

Interfaces are a special case of abstract classes in which **all methods** are abstract. Using Interfaces, specification and implementation of functionality can be kept apart. In our cruise example we have some ships supporting satellite TV and some who don't. The ships who do, have the methods `enableTV()` and `disableTV()`. It is useful to define an interface for that:

PHP Code:

```
interface SatelliteTV {
    public function enableTV();
    public function disableTV();
```

(continues on next page)

(continued from previous page)

```
}

class Luxusliner extends Ship implements SatelliteTV {

   protected $tvEnabled = FALSE;

   public function enableTV() {
          $this->tvEnabled = TRUE;
   }
   public function disableTV() {
          $this->tvEnabled = FALSE;
   }
}
```

Using the key word `implements` it is made sure, that the class implements the given interface. All methods in the interface definition then have to be realized. The object `LuxuryLiner` now is of the type `Ship` but also of the type `SatelliteTV`. It is also possible to implement not only one interface class but multiple, separated by comma. Of course interfaces can also be inherited by other interfaces.

Visibilities: public, private and protected

Access to properties and methods can be restricted by different visibilities to hide implementation details of a class. The meaning of a class can be communicated better like this, for implementation details in internal methods can not be accessed from outside. The following visibilities exist:

- **public**: properties and methods with this visibility can be accessed from outside the object. If no Visibility is defined, the behavior of `public` is used.
- **protected**: properties and methods with visibility `protected` can only be accessed from inside the class and it's child classes.
- **private**: properties and methods set to `private` can only be accessed from inside the class itself, not from child classes.

Access to Properties

This small example demonstrates how to work with protected properties:

PHP Code:

```
abstract class Ship {
   protected $coaches;
   ...
   abstract protected function setupCoaches();
}

class Luxusliner extends Ship {
   protected function setupCoaches() {
          $this->coaches = 300;
   }
}

$luxusliner = new Luxusliner('Fidelio', 100);
echo 'Number of coaches: ' . $luxusliner->coaches; // Does NOT work!
```

The `LuxuryLiner` may alter the property `coaches`, for this is `protected`. If it was `private` no access from inside of the child class would be possible. Access from outside of the hierarchy of inheritance (like in the last line of the example) is not possible. It would only be possible if the property was `public`.

We recommend to define all properties as `protected`. Like that, they can not be altered any more from outside and you should use special methods (called getter and setter) to alter or read them. We'll explain the use of these methods in the following section.

Access to Methods

All methods the object makes available to the outside have to be defined as `public`. All methods containing implementation details, e.g. `setupCoaches()` in the above example, should be defined as `protected`. The visibility `private` should be used most rarely, for it prevents methods from being overwritten or extended.

Often you'll have to read or set properties of an object from outside. So you'll need special methods that are able to set or get a property. These methods are called **setter** respectively **getter**. See the example.

PHP Code:

```
class Ship {

   protected $coaches;
   protected $classification = 'NORMAL';

   public function getCoaches() {
          return $this->coaches;
   }

   public function setCoaches($numberOfCoaches) {
          if ($numberOfCoaches > 500) {
                 $this->classification = 'LARGE';
          } else {
                 $this->classification = 'NORMAL';
          }
          $this->coaches = $numberOfCoaches;
   }

   public function getClassification() {
          return $this->classification;
   }

   ...
}
```

We now have a method `setCoaches()` which sets the number of coaches. Furthermore it changes - depending on the number of coaches - the ship category. You now see the advantage: When using methods to get and set the properties, you can perform more complex operations, as e.g. setting of dependent properties. This preserves consistency of the object. If you set `$coaches` and `$classification` to `public`, we could set the number of cabins to 1000 and classification to `NORMAL` - and our ship would end up being inconsistent.

Note: In Flow you'll find getter and setter methods all over. No property in Flow is set to `public`.

Static Methods and Properties

Until now we worked with objects, instantiated from classes. Sometimes though, it does not make sense to generate a complete object, just to be able to use a function of a class. For this php offers the possibility to directly access properties and methods. These are then referred to as `static properties` respectively `static methods`. Take as a rule of thumb: static properties are necessary, every time two instances of a class are to have a common property. Static methods are often used for function libraries.

Transferred to our example this means, that all ships are constructed by the same shipyard. in case of technical emergency, all ships need to know the actual emergency phone number of this shipyard. So we save this number in a static property `$shipyardSupportTelephoneNumber`:

PHP Code:

```
class Luxusliner extends Ship {
   protected static $shipyardSupportTelephoneNumber = '+49 30 123456';

   public function reportTechnicalProblem() {
          echo 'On the ship ' . $this->name . ' a problem has been discovered.
                Please inform ' . self::$shipyardSupportTelephoneNumber;
   }

   public static function setShipyardSupportTelephoneNumber($newNumber) {
          self::$shipyardSupportTelephoneNumber = $newNumber;
   }
}

$fidelio = new Luxusliner('Fidelio', 100);
$figaro = new Luxusliner('Figaro', 200);

$fidelio->reportTechnicalProblem();
$figaro->reportTechnicalProblem();

Luxusliner::setShipyardSupportTelephoneNumber('+01 1000');

$fidelio->reportTechnicalProblem();
$figaro->reportTechnicalProblem();

// Output
On the ship Fidelio a problem has been discovered. Please inform +49 30 123456
On the ship Figaro a problem has been discovered. Please inform +49 30 123456
On the ship Fidelio a problem has been discovered. Please inform +01 1000
On the ship Figaro a problem has been discovered. Please inform +01 1000
```

What happens here? We instantiate two different ships, which both have a problem and do contact the shipyard. Inside the method `reportTechnicalProblem()` you see that if you want to use static properties, you have to trigger them with the key word `self::`. If the emergency phone number now changes, the shipyard has to tell all the ships about the new number. For this it uses the **static method** `setShipyardSupportTelephoneNumber($newNumber)`. For the method is static, it is called through the scheme `classname::methodname()`, in our case `LuxuryLiner::setShipyardSupportTelephoneNumber(...)`. If you check the latter two problem reports, you see that all instances of the class use the new phone number. So both ship objects have access to the same static variable `$shipyardSupportTelephoneNumber`.

Important design- and architectural patterns

In software engineering you'll sooner or later stumble upon design problems that are connatural and solved in a similar way. Clever people thought about **design patterns** aiming to be a general solution to a problem. Each design pattern is so to speak a solution template for a specific problem. We by now have multiple design patterns that are successfully approved in practice and therefore have found there way in modern programming and especially Flow. In the following we don't want to focus on concrete implementation of the design patterns, for this knowledge is not necessary for the usage of Flow. Nevertheless deeper knowledge in design patterns in general is indispensable for modern programming style, so it might be fruitful for you to learn about them.

Tip: Further information about design patterns can e.g. be found on http://sourcemaking.com/ or in the book **PHP Design Patterns** by Stephan Schmidt, published by O'Reilly.

From the big number of design patterns, we will have a closer look on two that are essential when programming with Flow: **Singleton** & **Prototype**.

Singleton

This design pattern makes sure, that only one instance of a class can exist **at a time**. In Flow you can mark a class as singleton by annotating it with `@Flow\Scope("singleton")`. An example: our luxury liners are all constructed in the same shipyard. So there is no sense in having more than one instance of the shipyard object:

PHP Code:

```
/**
 * @Flow\Scope("singleton")
 */
class LuxuslinerShipyard {
   protected $numberOfShipsBuilt = 0;

   public function getNumberOfShipsBuilt() {
          return $this->numberOfShipsBuilt;
   }

   public function buildShip() {
          $this->numberOfShipsBuilt++;
          // Schiff bauen und zurückgeben
   }
}

$luxuslinerShipyard = new LuxuslinerShipyard();
$luxuslinerShipyard->buildShip();

$theSameLuxuslinerShipyard = new LuxuslinerShipyard();
$theSameLuxuslinerShipyard->buildShip();

echo $luxuslinerShipyard->getNumberOfShipsBuilt(); // 2
echo $theSameLuxuslinerShipyard->getNumberOfShipsBuilt(); // 2
```

Prototype

Prototype is sort of the antagonist to Singleton. While for each class only one object is instantiated when using Singleton, it is explicitly allowed to have multiple instances when using Prototype. Each class annotated with `@Flow\Scope("prototype")` is of type **Prototype**. Since this is the default scope, you can safely leave this one out.

Note: Originally for the design pattern **Prototype** is specified, that a new object is to be created by cloning an object prototype. We use Prototype as counterpart to Singleton, without a concrete pattern implementation in the background, though. For the functionality we experience, this does not make any difference: We invariably get back a new instance of a class.

Now that we refreshed your knowledge of object oriented programming, we can take a look at the deeper concepts of Flow: Domain Driven Design, Model View Controller and Test Driven Development. You'll spot the basics we just talked about in the following frequently.

2.1.3 Essential Design Patterns

Flow Paradigm

Flow was designed from the ground up to be modular, adaptive and agile to enable developers of all skill levels to build maintainable, extensible and robust software through the implementation of several proven design paradigms. Building software based on these principles will allow for faster, better performing applications that can be extended to meet changing requirements while avoiding inherent problems introduced by traditional legacy code maintenance. Flow aims to make what you "should" do what you "want" to do by providing the framework and community around best practices in the respective essential design patterns.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm which complements Object-Oriented Programming (OOP) by separating concerns of a software application to improve modularization. The separation of concerns (SoC) aims for making a software easier to maintain by grouping features and behavior into manageable parts which all have a specific purpose and business to take care of.

OOP already allows for modularizing concerns into distinct methods, classes and packages. However, some concerns are difficult to place as they cross the boundaries of classes and even packages. One example for such a cross-cutting concern is security: Although the main purpose of a Forum package is to display and manage posts of a forum, it has to implement some kind of security to assert that only moderators can approve or delete posts. And many more packages need a similar functionality for protect the creation, deletion and update of records. AOP enables you to move the security (or any other) aspect into its own package and leave the other objects with clear responsibilities, probably not implementing any security themselves.

Tip: Planning out the purpose and use cases of a package before you create it will allow for backwards compatibility by creating an unchanging interface for independent classes to consume.

Dependency Injection

In AOP there is focus on building reusable components that can be wired together to create a cohesive architecture. This goal becomes increasingly difficult because as the size and complexity of an application expands, so does its dependencies. One technique to aliviate dependency management is through Dependency Injection (DI).

Dependency Injection (DI) is a technique by which a package can request and gain access to another package simply by asking the injector. An injector is the service provided within a framework to instantiate and provide access to package interfaces upon request.

DI enables a package to control what dependencies it requires while allowing the framework or another third party system to handle the fullfillment of each dependency. This is know as Inversion of Control (IoC). IoC delegates the responsibility of dependency resolution to the framework while each package specifies which dependencies it needs.

AOP provides a means for interaction between packages through various interfaces and aspect. Without Dependency Injection AOP would suffer from creating untestable code by requiring you to manage dependencies in the constructor and thus breaking the Law of Demeter by allowing a package to "look" for its dependencies with a system instead of "asking" for them through the autonomous injector.

Test Driven Development

Test Driven Development (TDD) is a means in which a developer can explore, implement and verify various independent pieces of an application in order to deliver stable and maintainable code. TDD has become popular in mainstream development because the first step required is to think about what the purpose of a class or method is in the scope of your package's feature requirements incrementally, revising and refining small pieces of code while maintaining overall integrity of the system as whole.

Five Steps of Test Driven Development

1. **Think**: Before you write anything, consider what is required of the code you are about to create.
2. **Frame**: Write the simplest test possible, less than five lines of code or so that describe what you expect the method to do.
3. **Fulfill**: Again, write a small amount of code to meet the expectations of your test so that is passes. (It's acceptable to hard code variables and returns as you explore and think about the method, cleaning it up as you go.)
4. **Re-factor**: Now that you have a simple passing test, you know that your code as it stands works and can work on making it better while keeping an eye on if it breaks of not. Think about ways to improve your code by removing duplication and other "ugly" code until you feel it looks correct. Re-run the tests and make sure it still passes, if not, fix it.
5. **Repeat**: Do it again. Look at your test to make sure you are testing what it should do, not what it is doing. Add to your test if you find something missing and continue looping through the process until you're happy that the code can't be made any clearer with its current set of requirements. The more times you repeat, the better the resulting code will be.

Domain Driven Design

Domain-driven Design (DDD) is a practice where an implementation is deeply coupled with the evolving business model within its respective domain. Typically when working with DDD, technical experts are paired with a domain experts to ensure that each iteration of a system is getting closer to the core problem.

DDD relies on the following foundational elements:

- **Domain**: An ontology of concepts related to a specific area of knowledge and information.
- **Model**: An abstract system that describes the various aspects of a domain.
- **Ubiquitous Language**: A glossary of language structured around a domain model to connect all aspects of a model with uniformed definitions.
- **Context**: The relative position in which an expression of words are located that determine it's overall meaning.

In DDD the Domain Model that is formed is a guide or measure of the overall implementation of an applications relationship to the core requirements of the problem it is trying to solve. DDD is not a specific technique or way of developing software, it is a system to ensure that the desired result and end result of a development iteration or aligned. For this reason, DDD is often coupled with TDD and AOP.

2.1.4 Domain-Driven Design

Domain-Driven Design is a development technique which focuses on understanding the customer's problem domain. It not only contains a set of technical ideas, but it also consists of techniques to structure the creativity in the development process.

The key of Domain-Driven Design is understanding the customers needs, and also the environment in which the customer works. The problem which the to-be-written program should solve is called the *problem domain*, and in Domain-Driven Design, development is guided by the exploration of the problem domain.

While talking to the customer to understand his needs and wishes, the developer creates a model which reflects the current understanding of the problem. This model is called *Domain Model* because it should accurately reflect the problem domain of the customer. Then, the domain model is tested with real use-cases, trying to understand if it fits to the customer's processes and way of working. Then, the model is refined again – and the whole process of discussion with the customer starts again. Thus, Domain-Driven Design is an iterative approach to software development.

Still, Domain-Driven Design is very pragmatic, as code is created very early on (instead of extensive requirements specifications); and real-world problems thus occur very early in the development process, where they can be easily corrected. Normally, it takes some iterations of model refinement until a domain model adequately reflects the problem domain, focusing on the important properties, and leaving out unimportant ones.

In the following sections, some core components of Domain-Driven Design are explained. It starts with an approach to create a ubiquitous language, and then focuses on the technical realization of the domain model. After that, it is quickly explained how Flow enables Domain-Driven Design, such that the reader gets a more practical understanding of it.

Note: We do not explain all details of Domain-Driven Design in this work, as only parts of it are important for the general understanding needed for this work. More information can be found at [Evans].

Creating a Ubiquitous Language

In a typical enterprise software project, a multitude of different roles are involved: For instance, the customer is an expert in his business, and he wants to use software to solve a certain problem for him. Thus, he has a very clear idea on the interactions of the to-be-created software with the environment, and he is one of the people who need to use the software on a daily basis later on. Because he has much knowledge about how the software is used, we call him the *Domain Expert*.

On the other hand, there are the developers who actually need to implement the software. While they are very skilled in applying certain technologies, they often are no experts in the problem domain. Now, developers and domain experts speak a very different language, and misconceptions happen very often.

To reduce miscommunication, a *ubiquitous language* should be formed, in which key terms of the problem domain are described in a language understandable to both the domain expert and the developer. Thus, the developers learn to use the correct language of the problem domain right from the beginning, and can express themselves in a better way when discussing with the domain expert. Furthermore, they should also use the ubiquitous language throughout all parts of the project: Not only in communication, design documents and documentation, but the key terms should also appear in the domain model. Names of classes, methods and properties are also part of the ubiquitous language.

By using the language of the domain expert also in the code, it is possible to discuss about difficult-to-specify functionality by looking at the code together with the domain expert. This is especially helpful for complex calculations or difficult-to-specify condition rules. Thus, the domain expert can decide whether the business logic was correctly implemented.

Creating a ubiquitous language involves creating a glossary, in which the key terms are explained in a way both understandable to the domain expert and the developer. This glossary is also updated throughout the project, to reflect new insights gained in the development process.

Modelling the domain

Now, while discussing the problem with the domain expert, the developer starts to create the domain model, and refines it step by step. Usually, UML is employed for that, which just contains the relevant information of the problem domain.

The domain model consists of objects (as DDD is a technique for object-oriented languages), the so-called *Domain Objects*.

There are two types of domain objects, called *Entities* and *Value Objects*. If a domain object has a certain *identity* which stays the same as the objects changes its state, the object is an *entity*. Otherwise, if the identity of an object is only defined from *all properties*, it is a *value object*. We will now explain these two types of objects in detail, including practical use-cases.

Furthermore, association mapping is explained, and aggregates are introduced as a way to further structure the code.

Entities

Entities have a unique identity, which stays the same despite of changes in the properties of the object. For example, a user can have a user name as identity, a student a matriculation ID. Although properties of the objects can change over time (for example the student changes his courses), it is still the same object. Thus, the above examples are *entities*.

The identity of an object is given by an immutable property or a combination of them. In some use-cases it can make a lot of sense to define identity properties in a way which is *meaningful in the domain context*: If building an application which interfaces with a package tracking system, the tracking ID of a package should be used as identity inside the system. Doing so will reduce the risk of inconsistent data, and can also speed up access.

For some domain objects like a `Person`, it is highly dependent on the problem domain what should be used as identity property. In an internet forum, the e-mail address is often used as identity property for people, while when

implementing an e-government application, one might use the passport ID to uniquely identify citizens (which nobody would use in the web forum because its data is too sensitive).

In case the developer does not specify an identity property, the framework assigns a universally unique identifier (UUID) to the object at creation time.

It is important to stress that identity properties need to be set *at object creation time*, i.e. inside the constructor of an object, and are not allowed to change throughout the whole object lifetime. As we will see later, the object will be referenced using its identity properties, and a change of an identity property would effectively wipe one object and create a new one without updating dependent objects, leaving the system in an inconsistent state.

In a typical system, many domain objects will be *entities*. However, for some use-cases, another type is a lot better suited: Value objects, which are explained in the next section.

Value Objects

PHP provides several value types which it supports internally: Integer, float, string, float and array. However, it is often the case that you need more complex types of values inside your domain. These are being represented using *value objects*.

The identity of a value object is defined by *all its properties*. Thus, two objects are equal if all properties are equal. For instance, in a painting program, the concept of *color* needs to be somewhere implemented. A color is only represented through its value, for instance using RGB notation. If two colors have the same RGB values, they are effectively similar and do not need to be distinguished further.

Value objects do not only contain data, they can potentially contain very much logic, for example for converting the color value to another color space like HSV or CMYK, even taking color profiles into account.

As all properties of a value object are part of its identity, they are not allowed to be changed after the object's creation. Thus, value objects are *immutable*. The only way to "change" a value object is to create a new one using the old one as basis. For example, there might be a method `mix` on the `Color` object, which takes another `Color` object and mixes both colors. Still, as the internal state is not allowed to change, the mix method will effectively return a new `Color` object containing the mixed color values.

As value objects have a very straightforward semantic definition (similar to the simple data types in many programming languages), they can easily be created, cloned or transferred to other subsystems or other computers. Furthermore, it is clearly communicated that such objects are simple *values*.

Internally, frameworks can optimize the use of value objects by re-using them whenever possible, which can greatly reduce the amount of memory needed for applications.

Entity or Value Object?

An object can not be ultimately categorized into either being an entity or a value object – it depends greatly on the use case. An example illustrates this: For many applications which need to store an *address*, this address is clearly a value object - all properties like street, number, or city contribute to the identity of the object, and the *address* is only used as container for these properties.

However, if implementing an application for a postal service which should optimize letter delivery, not only the address, but also the person delivering to this location should be stored. This name of the postman does not belong to the identity of the object, and can change over time – a clear sign of *Address* being an entity in this case. So, generally it often depends on the use-case whether an object is an entity or value object.

People new to Domain-Driven Design often tend to overuse entities, as this is what people coming from a relational database background are used to.

So why not just use entities all the time? The design/architectural answer is: because a value object might just be more fitting your problem at hand. The technical answer is: because value objects are immutable and therefore avoid aliasing[1] problems, which are common cause of all kinds of bugs.

Associations

Now, after explaining the two types of domain objects, we will look at a particularly important implementation area: Associations between objects.

Domain objects have relationships between them. In the domain language, these relations are expressed often as follows: A *consists of* B, C *has* D, E *processes* F, G *belongs to* H. These relations are called *associations* in the domain model.

In the real world, relationships are often inherently bidirectional, are only active for a certain time span, and can contain further information. However, when modelling these relationships as associations, it is important to simplify them as much as possible, encoding only the relevant information into the domain model.

Especially complex to implement are bidirectional many-to-many relations, as they can be traversed in both directions, and consist of two lists of objects which have to be kept in sync manually in most programming languages (such as Java or PHP).

Still, especially in the first iterations of refining the domain model, many-to-many relations are very common. The following questions can help to simplify them:

- Is the association relevant for the core functionality of the application? If it is only used in rare use cases and there is another way to receive the needed information, it is often better to drop the association altogether.
- For bidirectional associations, can they be converted to unidirectional associations, because there is a main traversal direction? Traversing the other direction is still possible by querying the underlying persistence system.
- Can the association be qualified more restrictively, for example by adding multiplicities on each side?

The more simple the association is, the more directly it can be mapped to code, and the more clear the intent is.

Aggregates

When building a complex domain model, it will contain a lot of classes, all being on the same hierarchy level. However, often it is the case that certain objects are parts of a bigger object. For example, when modeling a `Car` domain object for a car repair shop, it might make sense to also model the wheels and the engine. As they are a part of the car, this understanding should be also reflected in our model.

Such a part-whole relationship of closely related objects is called *Aggregate*. An aggregate contains a root, the so-called *Aggregate Root*, which is responsible for the integrity of the child-objects. Furthermore, the whole aggregate has only one identity visible to the outside: The identity of the aggregate root object. Thus, objects outside of the aggregate are only allowed to persistently reference the aggregate root, and not one of the inner objects.

For the `Car` example this means that a `ServiceStation` object should not reference the engine directly, but instead reference the `Car` through its external identity. If it still needs access to the engine, it can retrieve it through the `Car` object.

These referencing rules effectively structure the domain model on a more fine-grained level, which reduces the complexity of the application.

[1] https://en.wikipedia.org/wiki/Aliasing_(computing)

Life cycle of objects

Objects in the real world have a certain life cycle. A car is built, then it changes during its lifetime, and in the end it is scrapped. In Domain-Driven Design, the life cycle of domain objects is very similar:

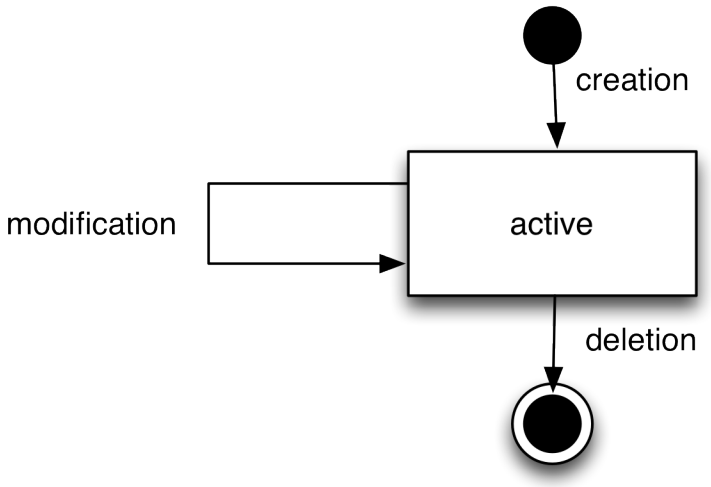

Fig. 1: Simplified life cycle of objects

Because of performance reasons, it is not feasible to keep all objects in memory forever. Some kind of persistent storage, like a database, is needed. Objects which are not needed at the current point in time should be persistently stored, and only transformed into objects when needed. Thus, we need to expand the `active` state from *Simplified life cycle of objects* to contain some more substates. These are shown below:

If an object is newly created, it is *transient*, so it is being deleted from memory at the end of the current request. If an object is needed permanently across requests, it needs to be transformed to a *persistent object*. This is the responsibility of *Repositories*, which allow to persistently store and retrieve domain objects.

So, if an object is *added* to a repository, this repository becomes responsible for saving the object. Furthermore, it is also responsible for persisting further changes to the object throughout its lifetime, automatically updating the database as needed.

For retrieving objects, repositories provide a query language. The repository automatically handles the database retrieval, and makes sure that each entity is only once in memory.

Despite the object being created and retrieved multiple times during its lifecycle, it logically continues to exist, even when it is stored in the database. It is only because of performance and safety reasons that is is not stored in main memory, but in a database. Thus, Domain-Driven Design distinguishes *creation* of an object from *reconstitution* from database: In the first case, the constructor is called, in the second case the constructor is not called as the object is only converted from another representation form.

In order to remove a persistent object, it needs to be removed from the repository responsible for it, and then at the end of the request, the object is transparently removed from the database.

For each *aggregate*, there is exactly one repository responsible which can be used to fetch the *aggregate root* object.

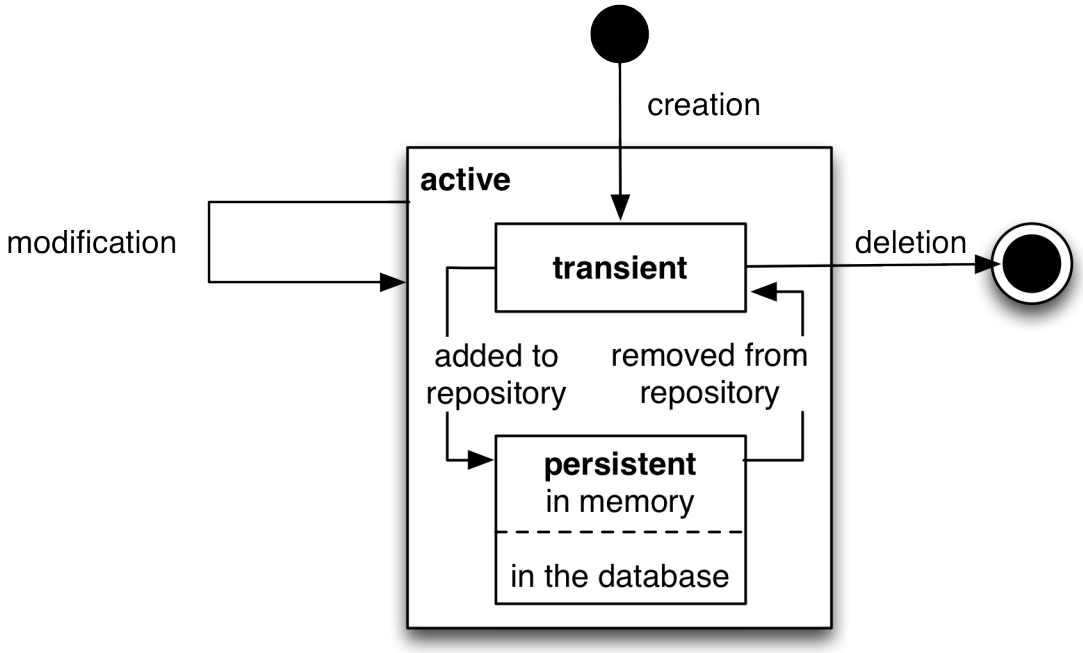

Fig. 2: The real life cycle of objects

How Flow enables Domain-Driven Design

Flow is a web development framework written in PHP, with Domain-Driven Design as its core principle. We will now show in what areas Flow supports Domain-Driven Design.

First, the developer can directly focus on creating the domain model, using unit testing to implement the use-cases needed. While he is creating the domain model, he can use plain PHP functionality, without caring about any particular framework. The PHP domain model he creates just consists of plain PHP objects, with no base class or other magic functionality involved. Thus, he can fully concentrate on domain modelling, without thinking about infrastructure yet.

This is a core principle of Flow: All parts of it strive for maximum focus and cleanness of the domain model, keeping the developer focused on the correct implementation of it.

Furthermore, the developer can use source code annotations to attach metadata to classes, methods or properties. This functionality can be used to mark objects as entity or value object, and to add validation rules to properties. In the domain object below, a sample of such an annotated class is given. As PHP does not have a language construct for annotations, this is emulated by Flow by parsing the source code comments.

In order to mark a domain object as *aggregate root*, only a repository has to be created for it, based on a certain naming convention. Repositories are the easiest way to make domain objects persistent, and Flow provides a base class containing generic `findBy*` methods. Furthermore, it supports a domain-specific language for building queries which can be used for more complex queries, as shown in below in the `AccountRepository`.

Now, this is all the developer needs to do in order to persistently store domain objects. The database tables are created automatically, and all objects get a UUID assigned (as we did not specify an identity property).

A simple domain object being marked as entity, and validation:

```
/**
 * @Flow\Entity
 */
class Account {

        /**
         * @var string
         */
        protected $firstName;

        /**
         * @var string
         */
        protected $lastName;

        /**
         * @var string
         * @Flow\Validate(type="EmailAddress")
         */
        protected $email;

        ... getters and setters as well as other functions ...
}
```

A simple repository:

```
class AccountRepository extends \Neos\Flow\Persistence\Repository {

                // by extending from the base repository, there is automatically a
                // findBy* method available for every property, i.e. findByFirstName(
↪"Sebastian")
                // will return all accounts with the first name "Sebastian".
        public function findByName($firstName, $lastName) {
                $query = $this->createQuery();
                $query->matching(
                        $query->logicalAnd(
                                $query->equals('firstName', $firstName),
                                $query->equals('lastName', $lastName)
                        )
                );
                return $query->execute();
        }
}
```

From the infrastructure perspective, Flow is structured as MVC framework, with the model being the Domain-Driven Design techniques. However, also in the controller and the view layer, the system has a strong support for domain objects: It can transparently convert objects to simple types, which can then be sent to the client's browser. It also works the other way around: Simple types will be converted to objects whenever possible, so the developer can deal with objects in an end-to-end fashion.

Furthermore, Flow has an Aspect-Oriented Programming framework at its core, which makes it easy to separate cross-cutting concerns. There is a security framework in place (built upon AOP) where the developer can declaratively define access rules for his domain objects, and these are enforced automatically, without any checks needed in the controller or the model.

There are a lot more features to show, like rapid prototyping support, dependency injection, a signal-slots system and a custom-built template engine, but all these should only aid the developer in focusing on the problem domain and writing decoupled and extensible code.

2.2 Part II: Getting Started

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

2.2.1 Introduction

What's Flow

Flow 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 Flow.

Needless to say, Flow 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.

What's in this tutorial?

This tutorial explains all the steps to get you started with your very own first Flow 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: 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 forum, as a ticket in our issue tracker or via Slack.

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

2.2.2 Requirements

Flow 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 Flow, because a server with exotic *php.ini* settings or wrong file permissions can easily spoil your day.

Server Environment

Not surprisingly, you'll need a web server for running your Flow-based web application. We recommend *Apache* (though *nginx*, *IIS* and others work too – we just haven't really tested them). Please make sure that the mod_rewrite module is enabled.

Tip: To enable Flow to create symlinks on Windows Server 2008 and higher you need to do some extra configuration. In IIS you need to configure *Authentication* for your site configuration to use a specific user in the *Anonymous Authentication* setting. The configured user should also be allowed to create symlinks using the local security policy *Local Policies* > *User Rights Assignments* > *Create symbolic links*

Flow's persistence mechanism requires a database supported by Doctrine DBAL. Make sure to use at least 10.2.2 for MariaDB, and 5.7.7 when using MySQL.

PHP

Flow was one of the first PHP projects taking advantage of namespaces and other features introduced in PHP version 5.3. By now we started using features of PHP 7.2, so make sure you have **PHP 7.2.0** or later available on your web server. Make sure your PHP CLI binary is the **same version**!

The default settings and extensions of the PHP distribution should work fine with Flow but it doesn't hurt checking if the PHP modules `mbstring`, `tokenizer` and `pdo_mysql` are enabled, especially if you compiled PHP yourself.

Note: Make sure the PHP functions `exec()`, `shell_exec()`, `escapeshellcmd()` and `escapeshellarg()` are not disabled in you PHP installation. They are required for the system to run.

The development context might 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.

In case you get a fatal error message saying something like `Maximum function nesting level of '100' reached, aborting!`, check your *php.ini* file for settings regarding Xdebug and modify/add a line `xdebug.max_nesting_level = 500` (suggested value).

2.2.3 Installation

Flow Download

Flow uses Composer for dependency management, which is a separate command line tool. Install it by following the installation instructions which boil down to this in the simplest case:

```
curl -s https://getcomposer.org/installer | php
```

Note: Feel free to install the composer command to a global location, by moving the phar archive to e.g. */usr/local/bin/composer* and making it executable. The following documentation assumes `composer` is installed globally.

Then use Composer in a directory which will be accessible by your web server to download and install all packages of the Flow Base Distribution. The following command will clone the latest stable version, include development dependencies and keep git metadata for future use:

```
composer create-project --keep-vcs neos/flow-base-distribution tutorial
```

This will install the latest stable version of Neos. In order to install a *specific version*, type:

```
composer create-project --keep-vcs neos/flow-base-distribution <target-directory>
↪<version>
```

And replace <*target-directory*> with the folder name to create the project in and <*version*> with the specific version to install, for example *1.2*. See [Composer documentation](https://getcomposer.org/doc/03-cli.md#create-project) for further details.

Note: Throughout this tutorial we assume that you installed the Flow 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.

To update all dependencies, run this from the top-level folder of the distribution:

```
composer update
```

Directory Structure

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

Directory	Description
Configuration/	Application specific configuration, grouped by contexts
Data/	Persistent and temporary data, including caches, logs, resources and the database
Packages/	Contains sub directories which in turn contain package directories
Packages/Framework/	Packages which are part of the official Flow distribution
Packages/Application/	Application specific packages
Packages/Libraries/	3rd party libraries
Web/	Public web root

A Flow application usually consists of the above directories. As you see, most of them contain data which is specific to your application, therefore upgrading the Flow distribution is a matter of updating *Packages/Framework/* and *Packages/Libraries/* when a new release is available.

Flow 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/*.

Basic Settings

In order to be able to run and serve out pages, Flow requires very few configurations. Flow uses so called `YAML` files for all it's configuration. If you don't know that yet, just take a look at the example, it is really easy to understand! For starters, you should begin by renaming the file `Configuration/Settings.yaml.example` to `Configuration/Settings.yaml`. This will be referenced elsewhere as the global settings file, because it lives in the installation directory, instead of a single package. It only contains the most basic configuration for a mysql database running on the same machine and a setting to enable the default Flow [routes](https://en.wikipedia.org/wiki/Web_framework#URL_mapping), which you need to see the "Welcome" page later.

```
Neos:
  Flow:
    persistence:
      backendOptions:
        driver: 'pdo_mysql'   # use pdo_pgsql for PostgreSQL
        charset: 'utf8mb4'    # change to utf8 when using PostgreSQL
        host: '127.0.0.1'     # adjust to your database host

    mvc:
      routes:
        'Neos.Flow': TRUE
```

Also, if you are trying this on Windows by chance, you need to uncomment the lines about the `phpBinaryPathAndFilename` and adjust the path to the `php.exe`. If you installed e.g. XAMPP, this should be `C:\path\to\xampp\php\php.exe`.

Other, more specific options should mostly only go directly into package specific `Settings.yaml` files. You will learn about those later.

File Permissions

Most of the directories and files must be readable and writable for the user you're running Flow with. This user will usually be the same one running your web server (`httpd`, `www`, `_www` or `www-data` on most Unix based systems). However it can and usually will happen that Flow 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 Flow must also be added this group. But don't worry, this is simply done by changing to the Flow base directory and calling the following command (this command must be called as super user):

```
sudo ./flow core:setfilepermissions john www-data www-data
```

Note:

> Setting file permissions is not necessary and not possible on Windows machines. For Apache to be able to create symlinks, you need to use Windows Vista (or newer) and Apache needs to be started with Administrator privileges. Alternatively

you can run the command `flow flow:cache:warmup` once from an Administrator elevated command line inside your installation folder. You then also need to repeat this step, whenever you install new packages.

Now that the file permissions are set, all users who plan using Flow 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-data john
```

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.

Web Server Configuration

As you have seen previously, Flow 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!

Configure AllowOverride and MultiViews

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

httpd.conf:

```
<Directory /var/apache2/htdocs/tutorial/>
        AllowOverride FileInfo Options=MultiViews
</Directory>
```

The way Flow addresses resources on the web makes it incompatible with the `MultiViews` feature of Apache. This needs to be turned off, the default `.htaccess` file distributed with Flow contains this code already

```
<IfModule mod_negotiation.c>

        # prevents Apache's automatic file negotiation, it breaks resource URLs
        Options -MultiViews

</IfModule>
```

Configure server-side scripts

Important: Disallow execution of server-side scripts below *Web/_Resources*. If users can upload (PHP) scripts they can otherwise be executed on the server. This should almost never be allowed, so make sure to disable PHP (or other script handlers) for anything below *Web/_Resources*.

The *.htaccess* file placed into the *Web/_Resources* folder does this for Apache when *.htaccess* is evaluated. Another way is to use this in the configuration:

```
<Directory /var/apache2/htdocs/tutorial/Web/_Resources>
        AllowOverride None
        SetHandler default-handler
        php_flag engine off
</Directory>
```

For nginx and other servers use similar configuration.

Configure a Context

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

Setting Up a Virtual Host for Context «Development»

Assuming that you chose Apache 2 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; make sure it is actually loaded with `Include` in `httpd.conf`):

httpd.conf:

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

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

Note: Flow runs per default in the `Development` context. That's why the *ServerName* in this example is **dev**.tutorial.local.

Setting Up a Virtual Host for Context «Production»

httpd.conf:

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

You'll be able to access the same application running in `Production` context by accessing the URL http://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:windowssystem32driversetchosts* on Windows):

hosts:

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

Change Context to «Production» without Virtual Host

If you decided to skip setting up virtual hosts earlier on, you can enable the `Production` context by editing the `.htaccess` file in the `Web` directory and remove the comment sign in front of the `SetEnv` line:

.htaccess:

```
# You can specify a default context by activating this option:
SetEnv FLOW_CONTEXT Production
```

Note: The concept of contexts and their benefits is explained in the next chapter «Configuration».

Welcome to Flow

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

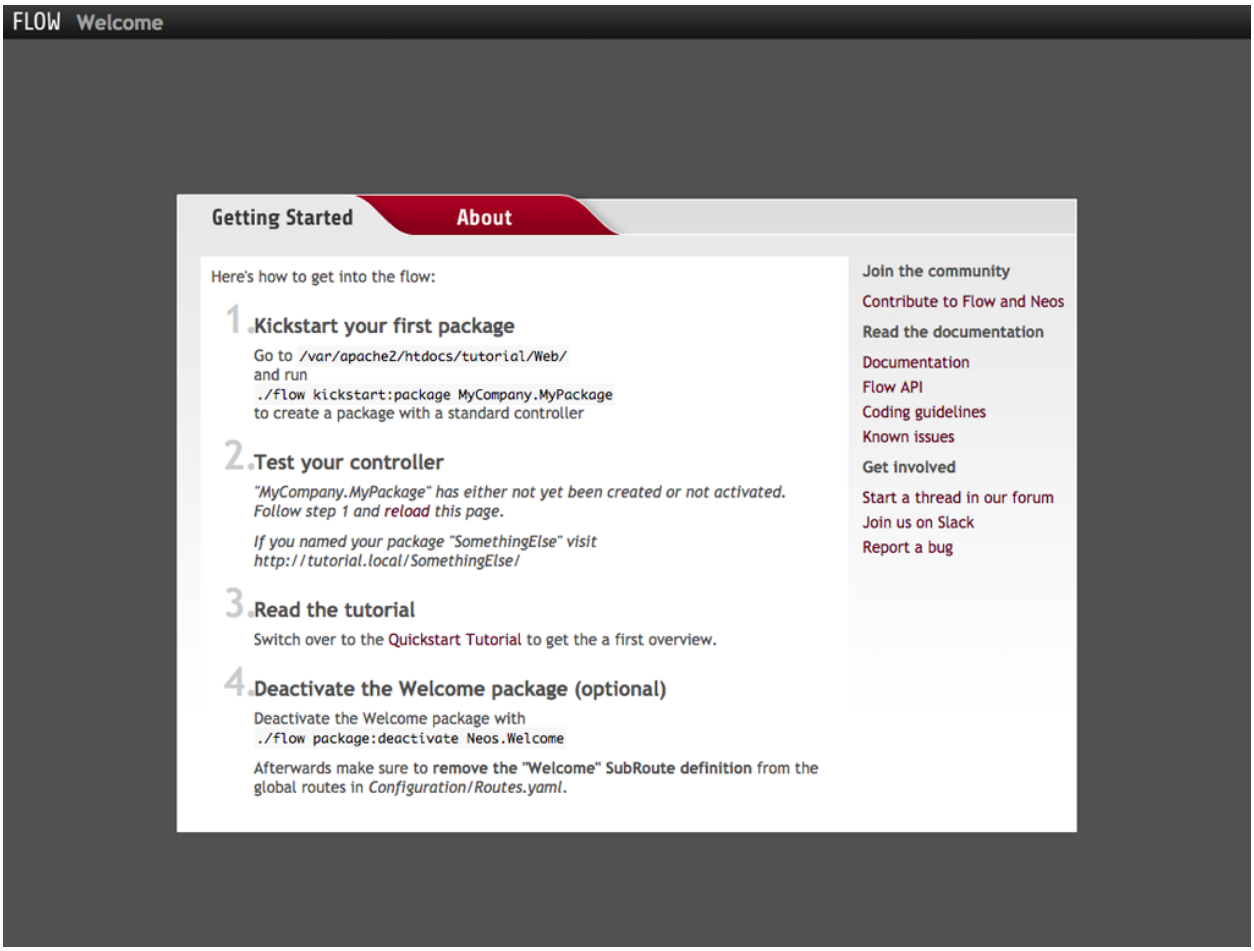

Fig. 3: The Flow Welcome screen

Tip: If you get in trouble during the installation ask for help at discuss.neos.io.

2.2.4 Configuration

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. Flow supports the notion of contexts which allow for bundling configuration for different purposes. Each Flow 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.

Flow provides configuration for the Production and Development context. 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).
- 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.

Tip: If Flow 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
Configuration/	Global configuration, for all contexts
Configuration/Development/	Configuration for the `Development` context
Configuration/Production/	Configuration for the `Production` context

Note: Setting Up Context with Virtual Host and change Context from «Development» to «Production» is explained

in the previous chapter «Installation».

One thing you certainly need to adjust is the database configuration. Aside from that Flow should work fine with the default configuration delivered with the distribution. However, there are many switches you can adjust: 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 Flow 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.

Flow uses the YAML format[1] for its configuration files. If you never edited a YAML file, there are two things you should know at least:

- Indentation has a meaning: by different levels of indentation, a structure is defined.
- Spaces, not tabs: you must indent with exactly 2 spaces per level, don't use tabs.

More detailed information about Flow's configuration management can be found in the Reference Manual.

Note: If you're running Flow 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 Flow caches files within the `Data/Temporary/<Context>/Caches/` directory whose absolute path can eventually become too long for Windows.

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 setting the `FLOW_PATH_TEMPORARY_BASE` environment variable - For example in the virtual host part of your Apache configuration:

httpd.conf:

```
<VirtualHost ...>
        SetEnv FLOW_PATH_TEMPORARY_BASE "C\\:tmp\\"
</VirtualHost>
```

Important: Parsing the YAML configuration files takes a bit of time which remarkably slows down the initialization of Flow. That's why all configuration is cached by default when Flow is running in Production context. Because this cache cannot be cleared automatically it is important to know that changes to any configuration file won't have any effect until you manually flush the respective caches.

To avoid any hassle we recommend that you stay in Development context throughout this tutorial.

Database Setup

Before you can store anything, you need to set up a database and tell Flow how to access it. The credentials and driver options need to be specified in the global Flow settings.

Tip: You should make it a habit to specify database settings in context-specific configuration files. This makes sure your functional tests will never accidentally truncate your production database. The same line of thought makes sense for other options as well, e.g. mail server settings.

[1] **YAML Ain't Markup Language** http://yaml.org

After you have created an empty database and set up a user with sufficient access rights, copy the file `Configuration/Development/Settings.yaml.example` to `Configuration/Development/Settings.yaml`. Open and adjust the file to your needs - for a common MySQL setup, it would look similar to this:

Configuration/Development/Settings.yaml:

```
Neos:
  Flow:
    persistence:
     backendOptions:
      dbname: 'gettingstarted'
      user: 'myuser'
      password: 'mypassword'
```

For global settings and Production context, the relevant files would be directly in `Configuration` respectively `Configuration/Production`.`

Tip: Configure your MySQL server to use the `utf8_unicode_ci` collation by default if possible!

If you configured everything correctly, the following command will create the initial table structure needed by Flow:

```
$ ./flow doctrine:migrate
Migrating up to 2011xxxxxxxxx from 0

++ migrating 20110613223837
        -> CREATE TABLE flow_resource_resourcepointer (hash VARCHAR(255) NOT NULL,␣
↪PRIMARY
        -> CREATE TABLE flow_resource_resource (persistence_object_identifier␣
↪VARCHAR(40)

...

  ------------------------

++ finished in 4.97
++ 5 migrations executed
++ 28 sql queries
```

Note: If you run into problems with the migrations, e.g. because the database does not allow dropping primary keys, there is another method to setup the database newly:

./flow doctrine:create && ./flow doctrine:migrationversion –add –version all

This should only be used for initial creation of the database, as it is a destructive operation though! Also note that this will not solve the issue for future migrations.

Environment Variables

Some specific flow behaviour can also be configured with a couple of environment variables.

Variable	Description
FLOW_ROOTPATH	Can be used to override the path to the Flow root.
FLOW_CONTEXT	Use to set the flow context (see above).
FLOW_PATH_TEMPORARY_BASE	Can be used to set a path for temporary data.
FLOW_LOCK_HOLDING_PAGE	Use to specify the html page shown when the site is locked. This is relative to the Packages directory. Can be given as FLOW_LOCKHOLDINGPAGE, too. That is deprecated as of Flow 8.0.
FLOW_ONLY_COMPOSER_LOADER	Set to true (1) to only use composer autoloader.

2.2.5 Modeling

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

We claim that Flow **lets you concentrate on the essential** and in fact this is one major design goal we followed in the making of Flow. 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. Flow 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.[1][2]

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 this 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 and 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[3] . 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 belong in this tutorial. A good starter is the *section about DDD* in the Flow documentation.

[1] http://en.wikipedia.org/wiki/Domain-driven_design

[2] 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.

[3] If it doesn't, we recommend reading our introductory sections about MVC in the *Flow reference*.

Domain Model

Our first Flow 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, what does our model look like? Our 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:

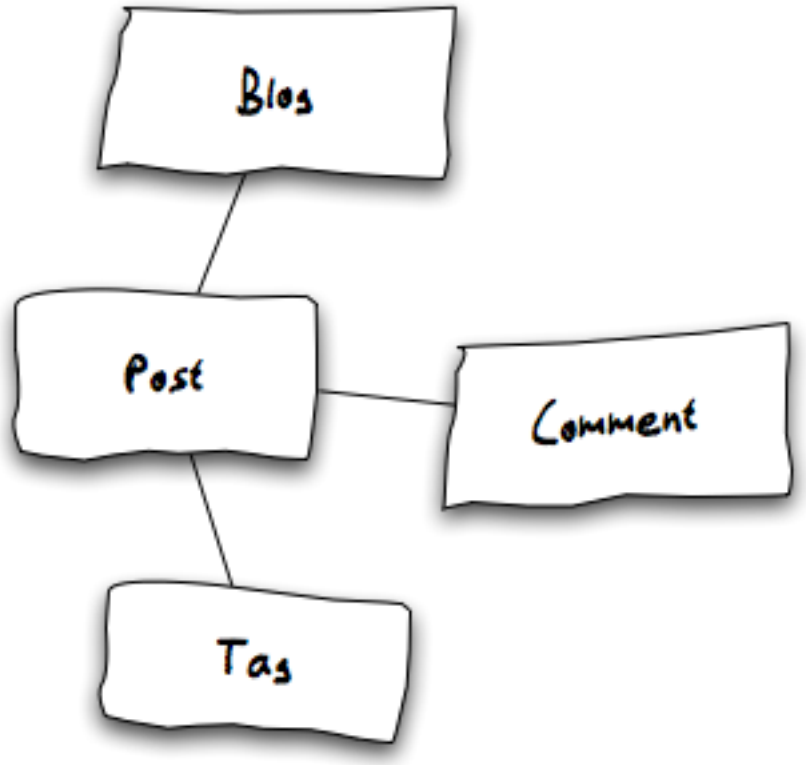

Fig. 4: A simple model

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

To be honest, the above model is not the best example of a rich Domain Model, compared to Active Records which usually contain not only properties but also methods.[4] 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.

[4] see http://en.wikipedia.org/wiki/Active_record_pattern

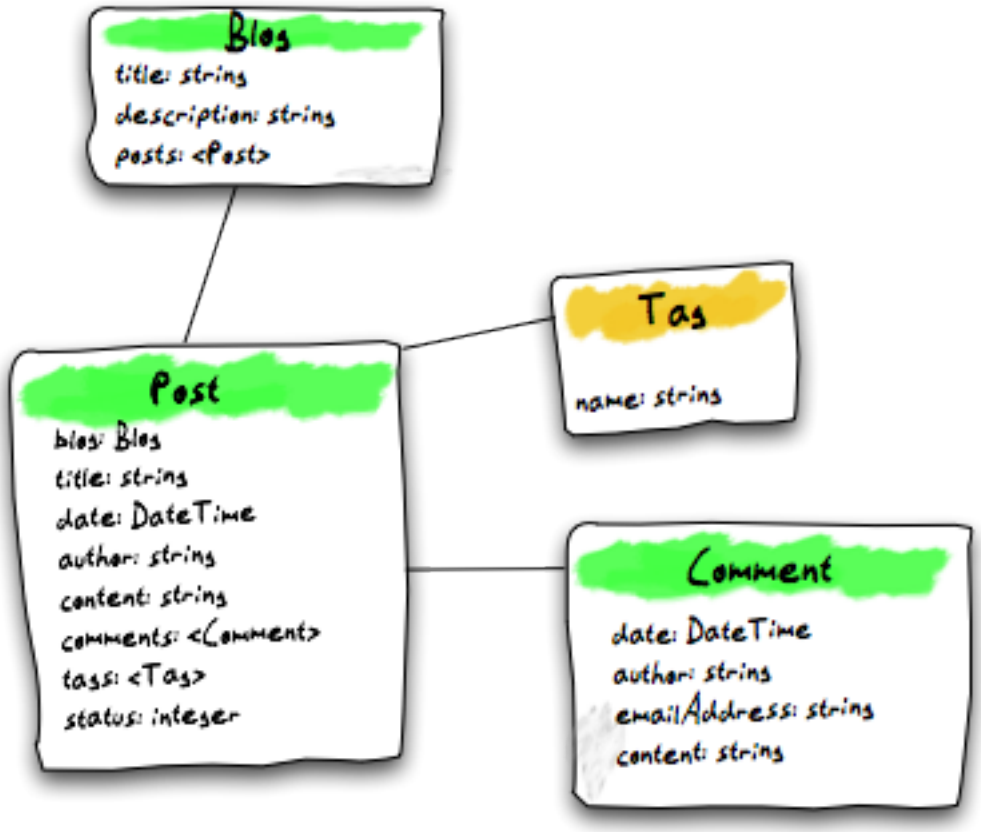

Fig. 5: Domain Model with properties

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[5] . Posts (i.e. `Post` objects) are stored in a `Blog` object in an array or better in an `Doctrine/Common/Collections/Collection`[6] instance. 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 the active Blog object?

All objects which can't be found by another object need to be stored in a repository. In Flow 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`:

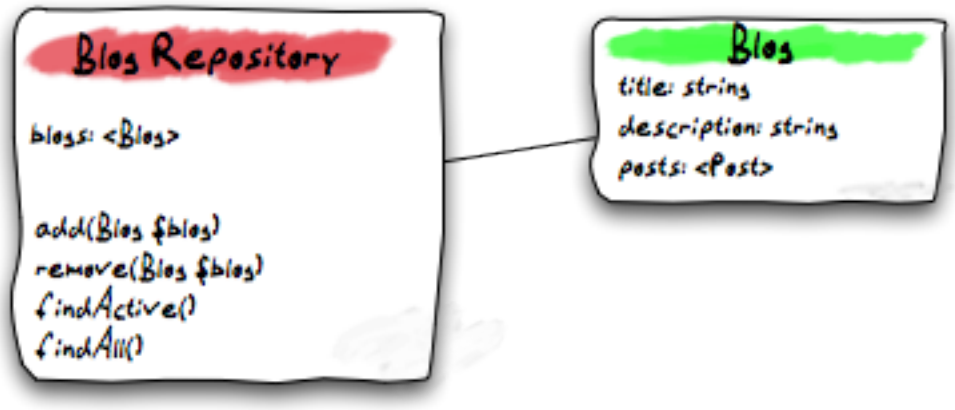

Fig. 6: Blog Repository and Blog

As you see, the `BlogRepository` provides methods for adding, removing and finding blogs. In our example application only one blog at a time is supported so all we need is a function to find the **active** blog – even though the repository can contain more than one blog.

Now, what if you want to display a list of the 5 latest 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:

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

[5] 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 loose the benefit of your IDE's autocompletion feature. Fortunately IDEs like Netbeans or Zend Studio provide functions to create getters and setters automatically.

[6] see http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html#collections

Fig. 7: A dedicated Post Repository

Aggregates

With the Post Repository 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**:

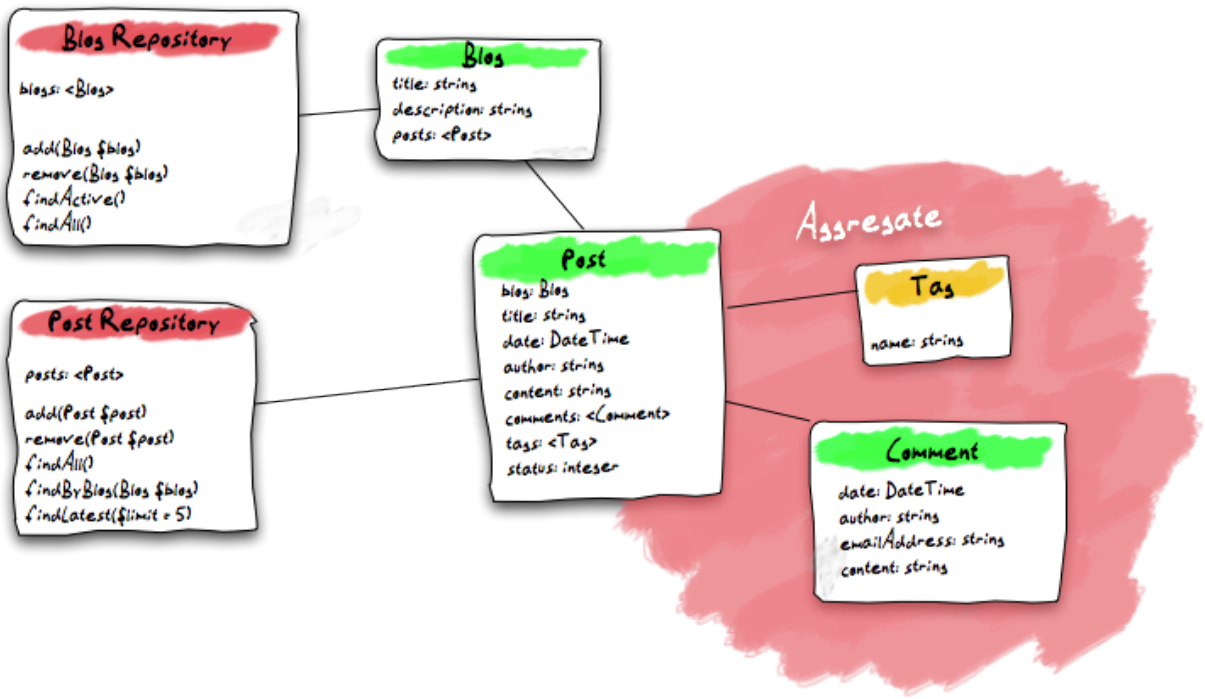

Fig. 8: The Post Aggregate

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 boundary* may exist.

Something to keep in mind is the opposite behavior the framework applies, when a repository for an object exists: any changes to it must be registered with that repository, as any persistence cascading of changes stops at aggregate boundaries.

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!

2.2.6 Kickstart

Flow makes it easy to start with a new application. The `Kickstarter` 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 Flow's command line interface. This might change in the future as a graphical interface to the kickstarter is developed.

Command Line Tool

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

```
./flow
Flow 3.0.0 ("Development" context)
usage: ./flow <command identifier>

See './flow help' for a list of all available commands.
```

To get an overview of all available commands, enter `./flow help`:

```
./flow help
Flow 3.0.0 ("Development" context)
usage: ./flow <command identifier>

The following commands are currently available:

PACKAGE "NEOS.FLOW":
------------------------------------------------------------------------------
* flow:cache:flush                          Flush all caches
  cache:warmup                              Warm up caches

  configuration:show                        Show the active configuration
                                            settings
  configuration:listtypes                   List registered configuration types
  configuration:validate                    Validate the given configuration
  configuration:generateschema              Generate a schema for the given
                                            configuration or YAML file.

* flow:core:setfilepermissions             Adjust file permissions for CLI and
                                            web server access
* flow:core:migrate                        Migrate source files as needed
* flow:core:shell                          Run the interactive Shell

  database:setcharset                       Convert the database schema to use
                                            the given character set and
                                            collation (defaults to utf8mb4 and
                                            utf8mb4_unicode_ci).

  doctrine:validate                         Validate the class/table mappings
  doctrine:create                           Create the database schema
  doctrine:update                           Update the database schema
  doctrine:entitystatus                     Show the current status of entities
                                            and mappings
  doctrine:dql                              Run arbitrary DQL and display
```

(continues on next page)

(continued from previous page)

```
                                        results
  doctrine:migrationstatus              Show the current migration status
  doctrine:migrate                      Migrate the database schema
  doctrine:migrationexecute             Execute a single migration
  doctrine:migrationversion             Mark/unmark a migration as migrated
  doctrine:migrationgenerate            Generate a new migration

  help                                  Display help for a command

  package:create                        Create a new package
  package:delete                        Delete an existing package
  package:activate                      Activate an available package
  package:deactivate                    Deactivate a package
  package:list                          List available packages
  package:freeze                        Freeze a package
  package:unfreeze                      Unfreeze a package
  package:refreeze                      Refreeze a package

  resource:publish                      Publish resources
  resource:clean                        Clean up resource registry

  routing:list                          List the known routes

  security:importpublickey              Import a public key
  security:importprivatekey             Import a private key
  security:showeffectivepolicy          Shows a list of all defined
                                        privilege targets and the effective
                                        permissions for the given groups.
  security:showunprotectedactions       Lists all public controller actions
                                        not covered by the active security
                                        policy
  security:showmethodsforprivilegetarget Shows the methods represented by the
                                        given security privilege target

  server:run                            Run a standalone development server

  typeconverter:list                    Lists all currently active and
                                        registered type converters


PACKAGE "NEOS.KICKSTARTER":
------------------------------------------------------------------------------
  kickstart:package                     Kickstart a new package
  kickstart:actioncontroller            Kickstart a new action controller
  kickstart:commandcontroller           Kickstart a new command controller
  kickstart:model                       Kickstart a new domain model
  kickstart:repository                  Kickstart a new domain repository

* = compile time command

See './flow help <commandidentifier>' for more information about a specific command.
```

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

Kickstart the package

Let's create a new package **Blog** inside the Vendor namespace **Acme**[1]:

```
./flow kickstart:package Acme.Blog
```

The kickstarter will create three files:

```
Created .../Acme.Blog/Classes/Controller/StandardController.php
Created .../Acme.Blog/Resources/Private/Layouts/Default.html
Created .../Acme.Blog/Resources/Private/Templates/Standard/Index.html
```

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

```
cd Packages/Application/
find Acme.Blog

Acme.Blog
Acme.Blog/Classes
Acme.Blog/Classes/Controller
Acme.Blog/Classes/Controller/StandardController.php
Acme.Blog/composer.json
Acme.Blog/Configuration
Acme.Blog/Documentation
Acme.Blog/Meta
Acme.Blog/Resources
Acme.Blog/Resources/Private
Acme.Blog/Resources/Private/Layouts
Acme.Blog/Resources/Private/Layouts/Default.html
Acme.Blog/Resources/Private/Templates
Acme.Blog/Resources/Private/Templates/Standard
Acme.Blog/Resources/Private/Templates/Standard/Index.html
Acme.Blog/Tests
Acme.Blog/Tests/Functional
Acme.Blog/Tests/Unit
```

Switch to your web browser and check at http://dev.tutorial.local/acme.blog if the generated controller produces some output:

Tip: If you get an error at this point, like a "404 Not Found" this could be caused by outdated cache entries. Because Flow should be running in `Development` context at this point, it is supposed to detect changes to code and resource files, but this seems to sometimes fail... Before you go crazy looking for an error on your side, **try reloading the page** and if that doesn't work you can **clear the cache manually** by executing the `./flow flow:cache:flush --force` command.

---

[1] A "vendor namespace" is used to avoid conflicts with other packages. It is common to use the name of the company/organization as namespace. See *Part III - Package Management* for some more information on package keys.

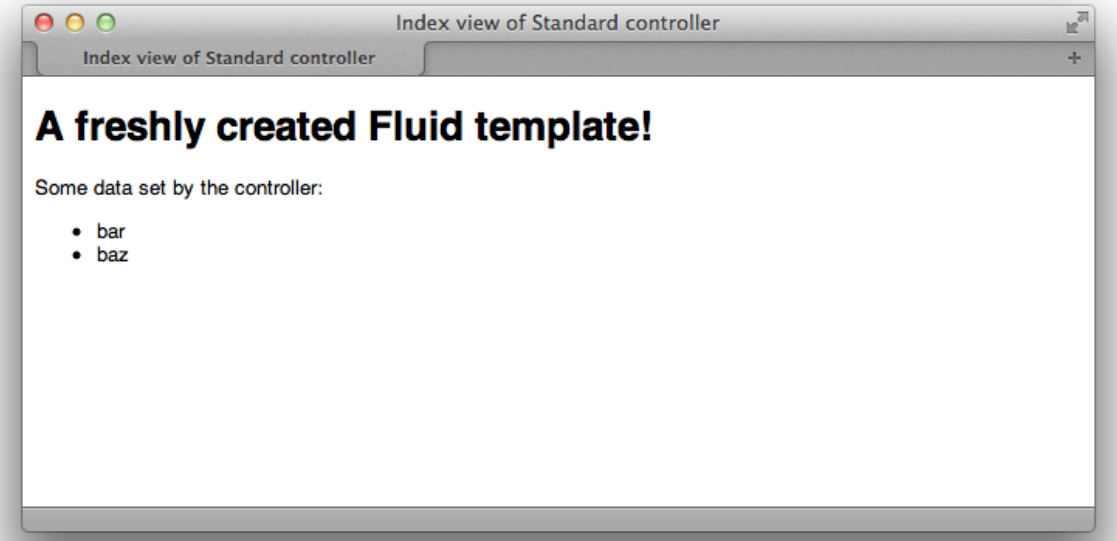

Fig. 9: A freshly created Fluid template

Kickstart Controllers

If you look at the drawing of our overall model you'll notice that you need controllers for the most important domain model, being `Post`. For the `PostController` we know that we'll need some standard actions, so let's have them created as well:

```
./flow kickstart:actioncontroller --generate-actions --generate-related Acme.Blog Post
```

resulting in:

```
Created .../Acme.Blog/Classes/Domain/Model/Post.php
Created .../Acme.Blog/Tests/Unit/Domain/Model/PostTest.php
Created .../Acme.Blog/Classes/Domain/Repository/PostRepository.php
Created .../Acme.Blog/Classes/Controller/PostController.php
Omitted .../Acme.Blog/Resources/Private/Layouts/Default.html
Created .../Acme.Blog/Resources/Private/Templates/Post/Index.html
Created .../Acme.Blog/Resources/Private/Templates/Post/New.html
Created .../Acme.Blog/Resources/Private/Templates/Post/Edit.html
Created .../Acme.Blog/Resources/Private/Templates/Post/Show.html
As new models were generated, don't forget to update the database schema with the␣
↪respective doctrine:* commands.
```

Tip: To see a full description of the kickstart commands and its options, you can display more details with `./flow help kickstart::actioncontroller`.

Once complete (in the Controller chapter), this new controller will be accessible via http://dev.tutorial.local/acme.blog/post

Please delete the file `StandardController.php` and its corresponding template directory as you won't need

them for our sample application[2].

Kickstart Models and Repositories

The kickstarter can also generate models and repositories, as you have seen above when using the `--generate-related` option while kickstarting the `PostController`. Of course that can also be done specifically with the `kickstart:model` command.

Before we do this, you should have a look at the next section on models and repositories.

2.2.7 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 Flow 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.

Tip: Code Examples

To see the full-scale code of the Blog as used by some of us, take a look at the Blog example package in our Git repository.

Domain models are really the heart of your application and therefore it is vital that this layer stays clean and legible. In a Flow application a model is just a plain old PHP object[1]. There's no need to write a schema definition, subclass a special base model or implement a required interface. All Flow requires from you as a specification for a model is a proper documented PHP class containing properties.

All your domain models need a place to live. 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. The base directory for the domain models is `Classes/<VendorName>/<PackageName>/Domain/Model/`.

Blog Model

The code for your `Blog` model can be kickstarted like this:

```
./flow kickstart:model Acme.Blog Blog title:string \
description:string 'posts:\Doctrine\Common\Collections\Collection'
```

That command will output the created file and a hint:

```
Created .../Acme.Blog/Classes/Acme/Blog/Domain/Model/Blog.php
Created .../Acme.Blog/Tests/Unit/Domain/Model/BlogTest.php
As a new model was generated, don't forget to update the database schema with the␣
↪respective doctrine:* commands.
```

Now let's open the generated `Blog.php` file and adjust it slightly:

- Add basic validation rules, see *Part III - Validation* for background information
- Add extended meta data for the ORM, see *Part III - Persistence*
- Replace the `setPosts()` setter by dedicated methods to add/remove single posts

[2] If you know you won't be using the StandardController, you can create a completely empty package with the `package:create` command.
[1] We love to call them POPOs, similar to POJOs http://en.wikipedia.org/wiki/Plain_Old_Java_Object

The resulting code should look like this:

Classes/Acme/Blog/Domain/Model/Blog.php:

```
<?php
namespace Acme\Blog\Domain\Model;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
use Neos\Flow\Annotations as Flow;
use Doctrine\ORM\Mapping as ORM;

/**
 * A blog that contains a list of posts
 *
 * @Flow\Entity
 */
class Blog {

        /**
         * @Flow\Validate(type="NotEmpty")
         * @Flow\Validate(type="StringLength", options={ "minimum"=3, "maximum"=80 })
         * @ORM\Column(length=80)
         * @var string
         */
        protected $title;

        /**
         * @Flow\Validate(type="StringLength", options={ "maximum"=150 })
         * @ORM\Column(length=150)
         * @var string
         */
        protected $description = '';

        /**
         * The posts contained in this blog
         *
         * @ORM\OneToMany(mappedBy="blog")
         * @ORM\OrderBy({"date" = "DESC"})
         * @var Collection<Post>
         */
        protected $posts;

        /**
         * @param string $title
         */
        public function __construct($title) {
                $this->posts = new ArrayCollection();
                $this->title = $title;
        }

        /**
         * @return string
```

(continues on next page)

(continued from previous page)

```
         */
        public function getTitle() {
                return $this->title;
        }

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

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

        /**
         * @param string $description
         * @return void
         */
        public function setDescription($description) {
                $this->description = $description;
        }

        /**
         * @return Collection
         */
        public function getPosts() {
                return $this->posts;
        }

        /**
         * Adds a post to this blog
         *
         * @param Post $post
         * @return void
         */
        public function addPost(Post $post) {
                $this->posts->add($post);
        }

        /**
         * Removes a post from this blog
         *
         * @param Post $post
         * @return void
         */
        public function removePost(Post $post) {
                $this->posts->removeElement($post);
        }

}
```

Tip: The *@Flow...* and *@ORM...* strings in the code are called *Annotations*. They are namespaced like PHP classes, so for the above code to work you **must** add a line like:

```
use Doctrine\ORM\Mapping as ORM;
```

to the files as well. Add it right after the *use* statement for the Flow annotations that is already there.

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:

Classes/Acme/Blog/Domain/Model/Blog.php:

```
namespace Acme\Blog\Domain\Model;
```

This namespace declaration must be the very first code in your file.

Classes/Acme/Blog/Domain/Model/Blog.php:

```
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
use Neos\Flow\Annotations as Flow;
use Doctrine\ORM\Mapping as ORM;
```

These *use* statements import PHP namespaces to the current scope. They are not required but can make the code much more readable. Look at the PHP manual to read more about PHP namespaces.

Classes/Acme/Blog/Domain/Model/Blog.php:

```
/**
 * A blog that contains a list of posts
 *
 * @Flow\Entity
 */
```

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

The 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 (i.e. stored in the database)
- According to DDD, an entity is an object which has an identity, that is even if two objects with the same values exist, their identity matters.

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

Classes/Acme/Blog/Domain/Model/Blog.php:

```
/**
 * @Flow\Validate(type="NotEmpty")
 * @Flow\Validate(type="StringLength", options={ "minimum"=3, "maximum"=80 })
 * @ORM\Column(length=80)
 * @var string
 */
protected $title;
```

(continues on next page)

(continued from previous page)

```
/**
 * @Flow\Validate(type="StringLength", options={ "maximum"=150 })
 * @ORM\Column(length=150)
 * @var string
 */
protected $description = '';

/**
 * The posts contained in this blog
 *
 * @ORM\OneToMany(mappedBy="blog")
 * @ORM\OrderBy({"date" = "DESC"})
 * @var Collection<Post>
 */
protected $posts;
```

Each property comes with a `@var` annotation which declares its type. Any type is fine, be it simple types (like `string`, `integer`, or `boolean`) or classes (like `\DateTime`, `\ACME\Foo\Domain\Model\Bar\Baz`, `Bar\Baz`, or an imported class like `Baz`).

The `@var` annotation of the `$posts` property differs a bit from the remaining comments when it comes to the type. This property holds a list of `Post` objects contained by this blog – in fact this could easily have been an array. However, an array does not allow the collection to be persisted by Doctrine 2 properly. We therefore use a `Collection`[2] instance (which is a `Doctrine\Common\Collections\Collection`, but we imported it to make the code more readable). The class name bracketed by the less-than and greater-than signs gives an important hint on the content of the collection (or array). There are a few situations in which Flow relies on this information.

The `OneToMany` annotation is Doctrine 2 specific and provides more detail on the type association a property represents. In this case it tells Doctrine that a `Blog` may be associated with many `Post` instances, but those in turn may only belong to one `Blog`. Furthermore the `mappedBy` attribute says the association is bidirectional and refers to the property `$blog` in the `Post` class.

The `OrderBy` annotation is regular Doctrine 2 functionality and makes sure the posts are always ordered by their date property when the collection is loaded.

The `Validate` annotations tell Flow about limits that it should enforce for a property. This annotation will be explained in the *Validation* chapter.

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 Flow - if you don't want to expose your properties it's fine to not define any setters or getters at all. The persistence framework can use other ways to access the properties' values.

Post Model

We need a model for the posts as well, so kickstart it like this:

```
./flow kickstart:model --force Acme.Blog Post \
        'blog:Blog' \
        title:string \
        date:\DateTime \
        author:string \
        content:string
```

Note that we use the `--force` option to overwrite the model - it was created along with the Post controller earlier because we used the `--generate-related` flag.

[2] https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html#collections

Adjust the generated code as follows:

Classes/Acme/Blog/Domain/Model/Post.php:

```
<?php
namespace Acme\Blog\Domain\Model;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Neos\Flow\Annotations as Flow;
use Doctrine\ORM\Mapping as ORM;

/**
 * @Flow\Entity
 */
class Post {

        /**
         * @Flow\Validate(type="NotEmpty")
         * @ORM\ManyToOne(inversedBy="posts")
         * @var Blog
         */
        protected $blog;

        /**
         * @Flow\Validate(type="NotEmpty")
         * @var string
         */
        protected $subject;

        /**
         * The creation date of this post (set in the constructor)
         *
         * @var \DateTime
         */
        protected $date;

        /**
         * @Flow\Validate(type="NotEmpty")
         * @var string
         */
        protected $author;

        /**
         * @Flow\Validate(type="NotEmpty")
         * @ORM\Column(type="text")
         * @var string
         */
        protected $content;

        /**
         * Constructs this post
         */
        public function __construct() {
                $this->date = new \DateTime();
```

(continues on next page)

(continued from previous page)

```
        }

        /**
         * @return Blog
         */
        public function getBlog() {
                return $this->blog;
        }

        /**
         * @param Blog $blog
         * @return void
         */
        public function setBlog(Blog $blog) {
                $this->blog = $blog;
                $this->blog->addPost($this);
        }

        /**
         * @return string
         */
        public function getSubject() {
                return $this->subject;
        }

        /**
         * @param string $subject
         * @return void
         */
        public function setSubject($subject) {
                $this->subject = $subject;
        }

        /**
         * @return \DateTime
         */
        public function getDate() {
                return $this->date;
        }

        /**
         * @param \DateTime $date
         * @return void
         */
        public function setDate(\DateTime $date) {
                $this->date = $date;
        }

        /**
         * @return string
         */
        public function getAuthor() {
                return $this->author;
        }

        /**
         * @param string $author
```

(continues on next page)

(continued from previous page)

```
         * @return void
         */
        public function setAuthor($author) {
                $this->author = $author;
        }

        /**
         * @return string
         */
        public function getContent() {
                return $this->content;
        }

        /**
         * @param string $content
         * @return void
         */
        public function setContent($content) {
                $this->content = $content;
        }

}
```

Blog Repository

According to our earlier statements regarding "Modeling", you need a repository for storing the blog:

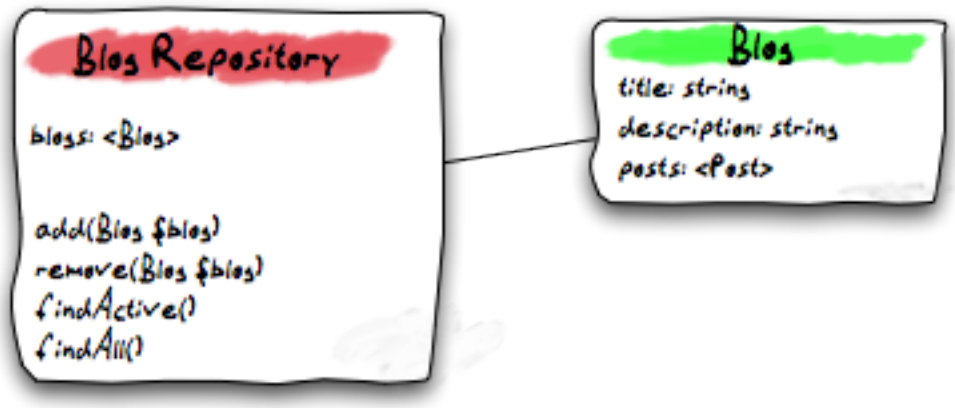

Fig. 10: Blog Repository and Blog

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 or controllers.

Similar to models the directory for your repositories is `Classes/Acme/Blog/Domain/Repository/`. You can kickstart the repository with:

```
./flow kickstart:repository Acme.Blog Blog
```

This will generate a vanilla repository for blogs containing this code:

Classes/Acme/Blog/Domain/Repository/BlogRepository.php:

```
<?php
namespace Acme\Blog\Domain\Repository;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Neos\Flow\Annotations as Flow;
use Neos\Flow\Persistence\Repository;

/**
 * @Flow\Scope("singleton")
 */
class BlogRepository extends Repository {

        // add customized methods here

}
```

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*`[3] methods. But for the sake of this demonstration lets assume we plan to have multiple blogs at some time. So lets add a `findActive()` method that - for now - just returns the first blog in the repository:

```
<?php
namespace Acme\Blog\Domain\Repository;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Acme\Blog\Domain\Model\Blog;
use Neos\Flow\Annotations as Flow;
use Neos\Flow\Persistence\Repository;

/**
 * @Flow\Scope("singleton")
 */
class BlogRepository extends Repository {

        /**
         * Finds the active blog.
         *
         * For now, only one Blog is supported anyway so we just assume that only one
         * Blog object resides in the Blog Repository.
         *
         * @return Blog The active blog or FALSE if none exists
         */
        public function findActive() {
```

(continues on next page)

[3] `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')`.

(continued from previous page)

```
                $query = $this->createQuery();
                return $query->execute()->getFirst();
        }

}
```

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` Flow assumes that it's supposed to store `Blog` objects.

To finish up, open the repository for our posts (which was generated along with the Post controller we kickstarted earlier) and add the following find methods to the generated code:

- `findByBlog()` to retrieve all posts of a given blog
- `findPrevious()` to get the previous post within the current blog
- `findNext()` to get the next post within the current blog

The resulting code should look like:

Classes/Acme/Blog/Domain/Repository/PostRepository.php:

```
<?php
namespace Acme\Blog\Domain\Repository;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Acme\Blog\Domain\Model\Blog;
use Acme\Blog\Domain\Model\Post;
use Neos\Flow\Annotations as Flow;
use Neos\Flow\Persistence\QueryInterface;
use Neos\Flow\Persistence\QueryResultInterface;
use Neos\Flow\Persistence\Repository;

/**
 * @Flow\Scope("singleton")
 */
class PostRepository extends Repository {

        /**
         * Finds posts by the specified blog
         *
         * @param Blog $blog The blog the post must refer to
         * @return QueryResultInterface The posts
         */
        public function findByBlog(Blog $blog) {
                $query = $this->createQuery();
                return
                        $query->matching(
                                $query->equals('blog', $blog)
                        )
                        ->setOrderings(array('date' => QueryInterface::ORDER_
↪DESCENDING))
                        ->execute();
```

(continues on next page)

(continued from previous page)

```
        }

        /**
         * Finds the previous of the given post
         *
         * @param Post $post The reference post
         * @return Post
         */
        public function findPrevious(Post $post) {
                $query = $this->createQuery();
                return
                        $query->matching(
                                $query->logicalAnd([
                                        $query->equals('blog', $post->getBlog()),
                                        $query->lessThan('date', $post->getDate())
                                ])
                        )
                        ->setOrderings(array('date' => QueryInterface::ORDER_
↪DESCENDING))
                        ->execute()
                        ->getFirst();
        }

        /**
         * Finds the post next to the given post
         *
         * @param Post $post The reference post
         * @return Post
         */
        public function findNext(Post $post) {
                $query = $this->createQuery();
                return
                        $query->matching(
                                $query->logicalAnd([
                                        $query->equals('blog', $post->getBlog()),
                                        $query->greaterThan('date', $post->getDate())
                                ])
                        )
                        ->setOrderings(array('date' => QueryInterface::ORDER_
↪ASCENDING))
                        ->execute()
                        ->getFirst();
        }

}
```

Tags and Comments

Until now, we have all the basics for our blog to function. A blog consists of multiple posts that each consists of a subject, content and some meta-data about the author and time of publishing. If you recall though, we also modelled the post to be labelled with one or multiple tags and users to comment on posts.

Without thinking, we might be starting to just copy & paste the code from the blog -> post relation, since that is also a 1:n relation. However, there are a few problems waiting for us, if we would go this route. Remember how we found that the comments and tags are parts of the `Post Aggregate`:

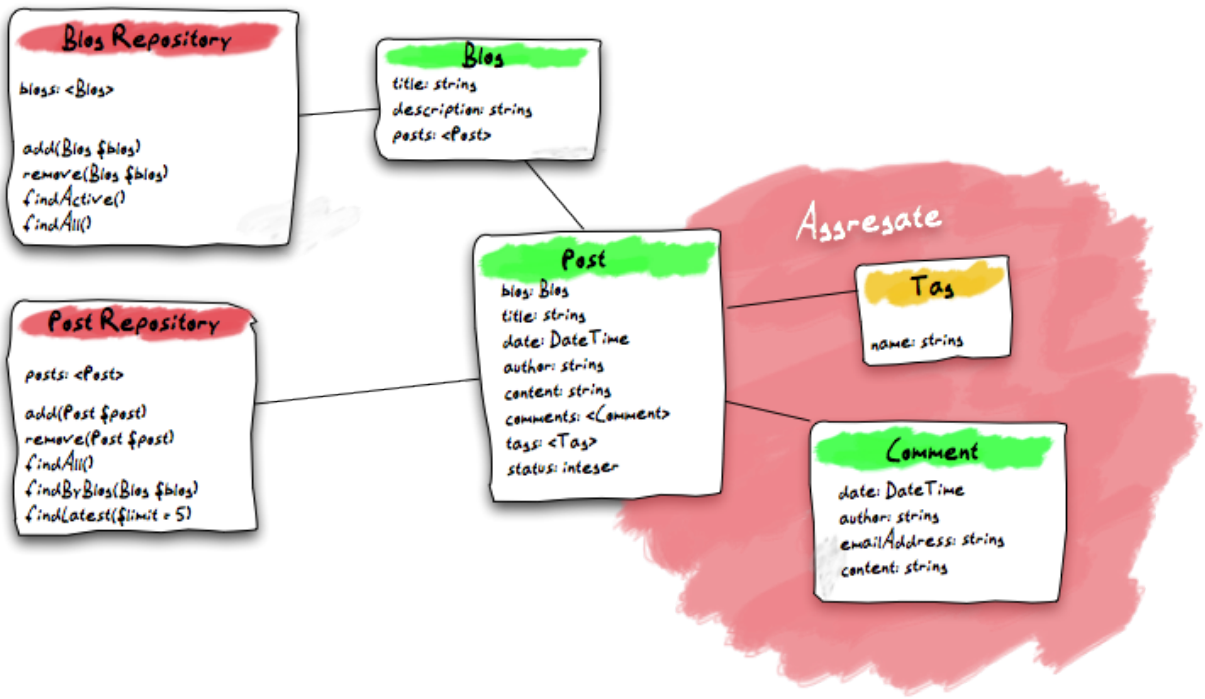

Fig. 11: The Post Aggregate

This means, that we should not have any means to directly access tags or comments outside of a post. Therefore they should not have a repository. Also, since we never directly access either one, there is no reason we need to reach the post they belong to. The access path is always in one direction starting from the post. In the terms of data modelling, we have a unidirectional one-to-many relation. To encode those in our domain model properly, unfortunately we have to go a non-obvious route. As we learned earlier, Doctrine provides a `OneToMany` annotation. This would seem like the right choice, but it isn't. `OneToMany` relations in Doctrine are always bidirectional and, even worse, the many side is the so called "owning side"[4] of the relation. This means, that to update the relation in any way, the owning side entity needs to be persisted. This is not matching our domain model, where the post is the `Aggregate Root` and hence the entity we persist from. To make Doctrine work as we intend our domain model, we need to annotate the relation as a `ManyToMany` and add a unique constraint on the "one" side[5]. In `ManyToMany` relations we can define the "owning side" freely and the additional constraint will prevent us from accidentally having more than one post referring to a specific comment.

First, let's add models for the comment and tag:

[4] https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/unitofwork-associations.html#bidirectional-associations
[5] https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html#one-to-many-unidirectional-with-join-table

```
./flow kickstart:model Acme.Blog Tag name:string
./flow kickstart:model Acme.Blog Comment \
        date:\DateTime \
        author:string \
        emailAddress:string \
        content:string
```

Then adjust the post model code as follows:

Classes/Acme/Blog/Domain/Model/Post.php:

```
<?php
namespace Acme\Blog\Domain\Model;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Neos\Flow\Annotations as Flow;
use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\Collection;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @Flow\Entity
 */
class Post {

        /**
         * @Flow\Validate(type="NotEmpty")
         * @ORM\ManyToOne(inversedBy="posts")
         * @var Blog
         */
        protected $blog;

        ...

        /**
         * @Flow\Validate(type="NotEmpty")
         * @ORM\Column(type="text")
         * @var string
         */
        protected $content;

        /**
         * @ORM\ManyToMany(orphanRemoval=true)
         * @ORM\JoinTable(inverseJoinColumns={@ORM\JoinColumn(unique=true)})
         * @var Collection<Comment>
         */
        protected $comments;

        /**
         * @ORM\ManyToMany
         * @var Collection<Tag>
         */
        protected $tags;
```

(continues on next page)

(continued from previous page)

```
        /**
         * Constructs this post
         */
        public function __construct() {
                $this->date = new \DateTime();
                $this->comments = new ArrayCollection();
                $this->tags = new ArrayCollection();
        }

        ...

        /**
         * @return Collection<Comment>
         */
        public function getComments() {
                return $this->comments;
        }

        /**
         * @param Comment $comment
         */
        public function addComment(Comment $comment) {
                $this->comments->add($comment);
        }

        /**
         * @param Comment $comment
         */
        public function deleteComment(Comment $comment) {
                $this->comments->remove($comment);
        }

        /**
         * @return Collection<Tag>
         */
        public function getTags() {
                return $this->tags;
        }

        /**
         * @param Tag $tag
         */
        public function addTag(Tag $tag) {
                $this->tags->add($tag);
        }

        /**
         * @param Tag $comment
         */
        public function removeTag(Tag $tag) {
                $this->tags->remove($tag);
        }
```

The `@ORM\JoinTable` annotation tells doctrine to enforce that each comment can only be referenced by one post. You might wonder why we have the `orphanRemoval=true` only on the comments, but not on the tags. `Orphan removal` tells doctrine to delete an entity, when the relation to it is unset, i.e. when the collections `remove()` method is invoked. Of course we do not want to delete a tag from the database completely, when we just untag a single

post, since another post might still have this tag.

2.2.8 Controller

Now that we have the first models and repositories in place we can almost move forward to creating our first controller. There are two types of controllers in Flow:

- `ActionControllers` are triggered by regular HTTP requests, and
- `CommandControllers` are usually invoked via the Command Line Interface.

Setup Controller

The `SetupCommandController` will be in charge of creating a `Blog` object, setting a title and description and storing it in the `BlogRepository`:

```
./flow kickstart:commandcontroller Acme.Blog Blog
```

The kickstarter created a very basic command controller containing only one command, the `exampleCommand`:

Classes/Acme/Blog/Command/BlogCommandController.php:

```
<?php
namespace Acme\Blog\Command;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Neos\Flow\Annotations as Flow;

/**
 * @Flow\Scope("singleton")
 */
class BlogCommandController extends \Neos\Flow\Cli\CommandController {

        /**
         * An example command
         *
         * The comment of this command method is also used for Flow's help screens.␣
↪The first line should give a very short
         * summary about what the command does. Then, after an empty line, you should␣
↪explain in more detail what the command
         * does. You might also give some usage example.
         *
         * It is important to document the parameters with param tags, because that␣
↪information will also appear in the help
         * screen.
         *
         * @param string $requiredArgument This argument is required
         * @param string $optionalArgument This argument is optional
         * @return void
         */
        public function exampleCommand($requiredArgument, $optionalArgument = NULL) {
```

(continues on next page)

(continued from previous page)

```
                $this->outputLine('You called the example command and passed "%s" as␣
↪the first argument.', array($requiredArgument));
        }

}
```

Let's replace the example with a `setupCommand` that can be used to create the first blog from the command line:

Classes/Acme/Blog/Command/BlogCommandController.php:

```
<?php
namespace Acme\Blog\Command;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Acme\Blog\Domain\Model\Blog;
use Acme\Blog\Domain\Model\Post;
use Acme\Blog\Domain\Repository\BlogRepository;
use Acme\Blog\Domain\Repository\PostRepository;
use Neos\Flow\Annotations as Flow;
use Neos\Flow\Cli\CommandController;

/**
 * @Flow\Scope("singleton")
 */
class BlogCommandController extends CommandController {

        /**
         * @Flow\Inject
         * @var BlogRepository
         */
        protected $blogRepository;

        /**
         * @Flow\Inject
         * @var PostRepository
         */
        protected $postRepository;

        /**
         * A command to setup a blog
         *
         * With this command you can kickstart a new blog.
         *
         * @param string $blogTitle the name of the blog to create
         * @param boolean $reset set this flag to remove all previously created blogs␣
↪and posts
         * @return void
         */
        public function setupCommand($blogTitle, $reset = FALSE) {
                if ($reset) {
                        $this->blogRepository->removeAll();
                        $this->postRepository->removeAll();
                }
```

(continues on next page)

(continued from previous page)

```
                $blog = new Blog($blogTitle);
                $blog->setDescription('A blog about Foo, Bar and Baz.');
                $this->blogRepository->add($blog);

                $post = new Post();
                $post->setBlog($blog);
                $post->setAuthor('John Doe');
                $post->setSubject('Example Post');
                $post->setContent('Lorem ipsum dolor sit amet, consectetur␣
↪adipisicing elit.' . chr(10) . 'Sed do eiusmod tempor incididunt ut labore et␣
↪dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco␣
↪laboris nisi ut aliquip ex ea commodo consequat.');
                $this->postRepository->add($post);

                $this->outputLine('Successfully created a blog "%s"', [$blogTitle]);
        }

}
```

You can probably figure out easily what the `setupCommand` does – it empties the `BlogRepository` and `PostRepository` if the `--reset` flag is set, creates a new `Blog` object and adds it to the `BlogRepository`. In addition a sample blog post is created and added to the `PostRepository` and blog. Note that if you had omitted the lines:

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

and

```
$this->postRepository->add($post);
```

the blog and the post would have been created in memory but not persisted to the database.

Using the blog and post repository sounds plausible, but where do you get the repositories from?

Classes/Acme/Blog/Command/BlogCommandController.php:

```
/**
 * @Flow\Inject
 * @var BlogRepository
 */
protected $blogRepository;
```

The property declarations for `$blogRepository` (and `$postRepository`) 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 Flow 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*[1], 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 on *objects* in *Part III: Manual* once you start with your own coding.

To create the required database tables we now use the command line support to generate the tables for our package:

[1] Remember, *prototype* is the default object scope and because the `BlogRepository` does contain a `Scope` annotation, it has the singleton scope instead.

```
./flow doctrine:migrationgenerate
```

```
Do you want to move the migration to one of these Packages?
  [0] Don't Move
  [1] Neos.Eel
  [2] Neos.Flow
  [3] Neos.Fluid
  [3] Neos.Kickstart
  [4] Neos.Welcome
  [5] Acme.Blog
```

Hit a key to move the new migration to the `Acme.Blog` package (in this example key "5") and press <ENTER>. You will now find the generated migration in *Migrations/Mysql/Version<YYYYMMDDhhmmss>.php*. Whenever you auto-generate a migration take a few minutes to verify that it contains (only) the changes you want to apply. In this case the migration should look like this:

```
<?php
namespace Neos\Flow\Persistence\Doctrine\Migrations;

use Doctrine\DBAL\Migrations\AbstractMigration,
        Doctrine\DBAL\Schema\Schema;

/**
 * Initial migration, creating tables for the "Blog" and "Post" domain models
 */
class Version20150714161019 extends AbstractMigration {

        /**
         * @param Schema $schema
         * @return void
         */
        public function up(Schema $schema) {
                $this->abortIf($this->connection->getDatabasePlatform()->getName() !=
↪"mysql");

                $this->addSql("CREATE TABLE acme_blog_domain_model_blog (persistence_
↪object_identifier VARCHAR(40) NOT NULL, title VARCHAR(80) NOT NULL, description␣
↪VARCHAR(150) NOT NULL, PRIMARY KEY(persistence_object_identifier)) DEFAULT␣
↪CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci ENGINE = InnoDB");
                $this->addSql("CREATE TABLE acme_blog_domain_model_post (persistence_
↪object_identifier VARCHAR(40) NOT NULL, blog VARCHAR(40) DEFAULT NULL, subject␣
↪VARCHAR(255) NOT NULL, date DATETIME NOT NULL, author VARCHAR(255) NOT NULL,␣
↪content LONGTEXT NOT NULL, INDEX IDX_EF2000AAC0155143 (blog), PRIMARY␣
↪KEY(persistence_object_identifier)) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_
↪unicode_ci ENGINE = InnoDB");
                $this->addSql("ALTER TABLE acme_blog_domain_model_post ADD CONSTRAINT␣
↪FK_EF2000AAC0155143 FOREIGN KEY (blog) REFERENCES acme_blog_domain_model_blog␣
↪(persistence_object_identifier)");
        }

        /**
         * @param Schema $schema
         * @return void
         */
        public function down(Schema $schema) {
                $this->abortIf($this->connection->getDatabasePlatform()->getName() !=
↪"mysql");
```

(continues on next page)

(continued from previous page)

```
                $this->addSql("ALTER TABLE acme_blog_domain_model_post DROP FOREIGN␣
↪KEY FK_EF2000AAC0155143");
                $this->addSql("DROP TABLE acme_blog_domain_model_blog");
                $this->addSql("DROP TABLE acme_blog_domain_model_post");
        }
}
```

Now you can execute all pending migrations to update the database schema:

```
./flow doctrine:migrate
```

And finally you can try out the `setupCommand`:

```
./flow blog:setup "My Blog"
```

and the CLI should respond with:

```
Successfully created a blog "My Blog"
```

This is all we need for moving on to something more visible: the blog posts.

Basic Post Controller

Now let us add some more code to *. . . /Classes/Acme/Blog/Controller/PostController.php*:

```
<?php
namespace Acme\Blog\Controller;

/*                                                                        *
 * This script belongs to the Flow package "Acme.Blog".                   *
 *                                                                        *
 *                                                                        */

use Acme\Blog\Domain\Repository\BlogRepository;
use Acme\Blog\Domain\Repository\PostRepository;
use Neos\Flow\Annotations as Flow;
use Neos\Flow\Mvc\Controller\ActionController;
use Acme\Blog\Domain\Model\Post;

class PostController extends ActionController {

        /**
         * @Flow\Inject
         * @var BlogRepository
         */
        protected $blogRepository;

        /**
         * @Flow\Inject
         * @var PostRepository
         */
        protected $postRepository;

        /**
```

(continues on next page)

(continued from previous page)

```
         * Index action
         *
         * @return string HTML code
         */
        public function indexAction() {
                $blog = $this->blogRepository->findActive();
                $output = '
                        <h1>Posts of "' . $blog->getTitle() . '"</h1>
                        <ol>';

                foreach ($blog->getPosts() as $post) {
                        $output .= '<li>' . $post->getSubject() . '</li>';
                }

                $output .= '</ol>';

                return $output;
        }

        // ...

}
```

The `indexAction` retrieves the active blog from the `BlogRepository` and outputs the blog's title and post subject lines[2]. A quick look at http://dev.tutorial.local/acme.blog/post[3] confirms that the `SetupController` has indeed created the blog and post:

Fig. 12: Output of the indexAction

[2] Don't worry, the action won't stay like this – of course later we'll move all HTML rendering code to a dedicated view.
[3] The *acme.blog* stands for the package *Acme.Blog* and *post* specifies the controller *PostController*.

Create Action

In the `SetupController` we have seen how a new blog and a post can be created and filled with some hardcoded values. At least the posts should, however, be filled with values provided by the blog author, so we need to pass the new post as an argument to a `createAction` in the `PostController`:

Classes/Acme/Blog/Controller/PostController.php:

```
// ...

/**
 * Creates a new post
 *
 * @param Post $newPost
 * @return void
 */
public function createAction(Post $newPost) {
        $this->postRepository->add($newPost);
        $this->addFlashMessage('Created a new post.');
        $this->redirect('index');
}
```

The `createAction` expects a parameter `$newPost` which is the `Post` object to be persisted. The code is quite straight-forward: add the post 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/acme.blog/post/create:

Flow analyzed the new method signature and automatically registered `$newPost` 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 post? You need to create some HTML form which allows you to enter the post details and then submits the information to the `createAction`. But you don't want the controller rendering such a form – this is clearly a task for the view!

2.2.9 View

The view's responsibility is solely the visual presentation of data provided by the controller. In Flow views are cleanly decoupled from the rest of the MVC framework. This allows you to either take advantage of Fluid (Flow'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 Flow'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.

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 Flow comes with a powerful resource manager 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 somewhere that is publicly accessible. By default, Flow just symlinks those files to the public resources directory below the `Web` folder.

Let's take a look at the directory layout of the *Acme.Blog* package:

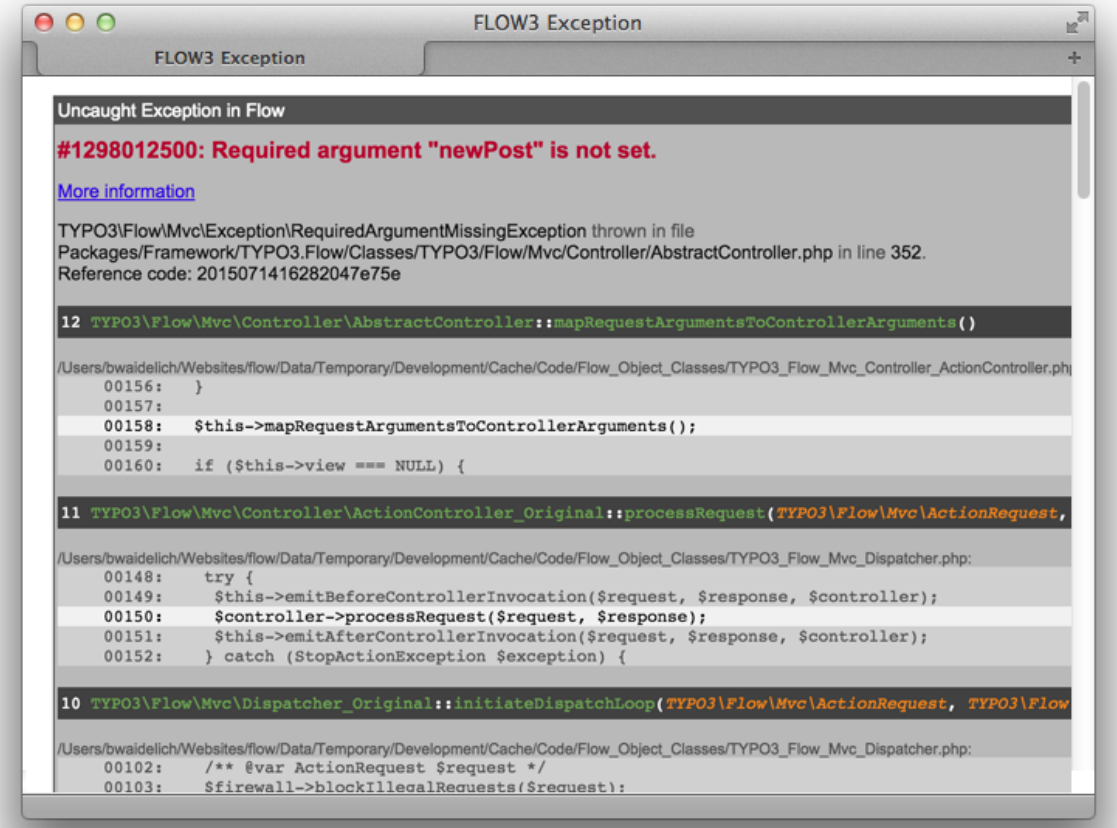

Fig. 13: Create action called without argument

Table 1: Directory structure of a Flow package

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

No matter what files and directories you create below `Resources/Public/` - all of them will, by default, be symlinked to `Web/_Resources/Static/Packages/Acme.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 *Part III*.

Important: For the blog example in this tutorial we created some style sheet to make it more appealing. If you'd like the examples to use those styles, then it's time to copy `Resources/Public/` from the git repository (https://github.com/neos/Acme.Blog) to your blog's public resources folder (`Packages/Application/Acme.Blog/Resources/Public/`).

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:

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 adjust the file called `Default.html` inside the `Acme.Blog/Resources/Private/Layouts` directory to contain the following code:

Resources/Private/Layouts/Default.html:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="utf-8">
    <title>{blog.title} - Flow Blog Example</title>
    <link rel="stylesheet" href="../../Public/Styles/App.css" type="text/css" />
</head>
<body>

    <header>
        <f:if condition="{blog}">
            <f:link.action action="index" controller="Post">
                <h1>{blog.title}</h1>
            </f:link.action>
            <p class="description">{blog.description -> f:format.crop(maxCharacters:␣
↪80)}</p>
        </f:if>
```

(continues on next page)

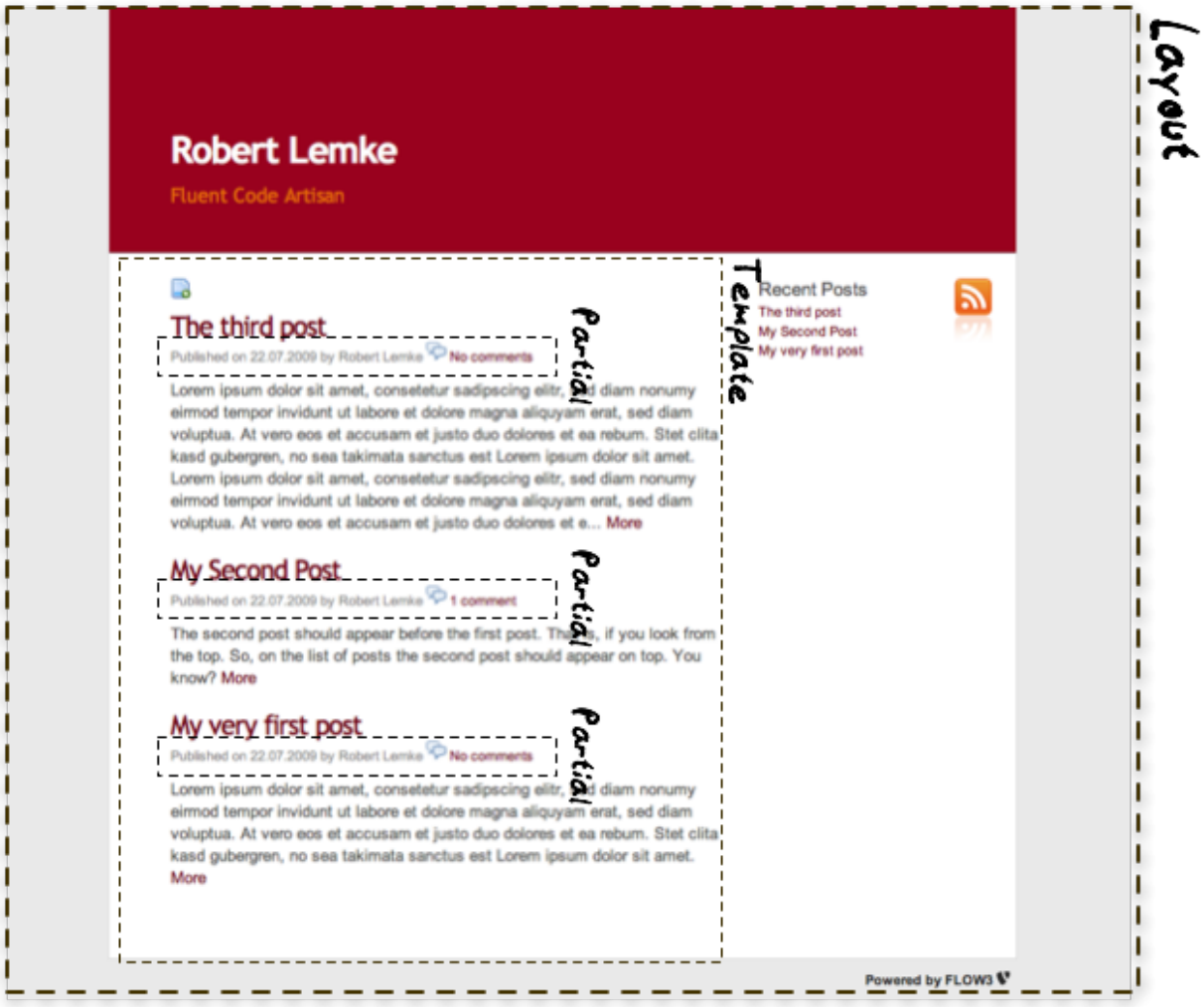

Fig. 14: Layout, Template and Partial

(continued from previous page)

```
    </header>

    <div id="content">
        <f:flashMessages class="flashmessages" />
        <f:render section="MainContent" />
    </div>

    <footer>
        <a href="http://flow.neos.io">
            Powered by Flow
        </a>
    </footer>

</body>
</html>
```

Tip: If you don't want to download the stylesheet mentioned above, you can import it directly from the github repository, replacing *../../Public/Styles/App.css* ` with `https://raw.githubusercontent.com/neos/Acme.Blog/master/Resources/Public/Styles/App.css` Of course you can also just remove the whole `<link rel="stylesheet" ...` line if you don't care about style.

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:if>` or `<f:link.action>`. 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 thing to notice is `<f:if>`, a Fluid tag in `<body>`. This tag instructs Fluid to render its content only if its condition is true. In this case, `condition="{blog}"` tells the `<f:if>` tag to render only if `blog` is set.

Look at that condition again, noting the curly braces: `{blog}`. This is a variable accessor. It is very similar to some Fluid markup that we skipped over in `<head>`:

Resources/Private/Layouts/Default.html:

```
<title>{blog.title} - Flow Blog Example</title>
```

As you will see in a minute, Fluid allows your controller to define variables for the template view. In order to display the blog's name, 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}` and Fluid will internally call the `getTitle()` method of the `Blog` instance.

We've looked at two kinds of Fluid syntax: tag-style view helpers (`<f:if>`), and variable accessors (`{blog.title}`). Another kind of Fluid syntax is an alternative way to address view helpers, the **view helper shorthand syntax**:

Resources/Private/Layouts/Default.html:

```
{blog.description -> f:format.crop(maxCharacters: 80)}
```

`{f:format.crop(...)}` instructs Fluid to crop the given value (in this case the Blog's description). With the `maxCharacters` argument the description will be truncated if it exceeds the given number of characters. The generated HTML code will look something like this:

Resources/Private/Layouts/Default.html:

```
This is a very long description that will be cropped if it exceeds eighty charac...
```

If you look at the remaining markup of the layout you'll find more uses of view helpers, including `flashMessages`. It generates an unordered list with all flash messages. Well, maybe you remember this line in the `createAction` of our `PostController`:

```
$this->addFlashMessage('Created a new post.');
```

Flash messages are a great way to display success or error messages to the user beyond a single request. And because they are so useful, Flow provides a `FlashMessageContainer` with some helper methods and Fluid offers the `flashMessages` view helper. Therefore, if you create a new post, you'll see the message *Your new post was created* at the top of your blog index on the next hit.

There's only one view helper you need to know about before proceeding with our first template, the **render** view helper:

Resources/Private/Layouts/Default.html:

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

This tag tells Fluid to insert the section `MainContent` 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 which layout it is based on, defines sections which in return are included by the layout. Confusing? Let's look at a concrete example.

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 Post controller's index action in `Acme.Blog/Resources/Private/Templates/Post/Index.html` with some more meaningful HTML:

Resources/Private/Templates/Post/Index.html:

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

<f:section name="MainContent">
    <f:if condition="{blog.posts}">
        <f:then>
            <ul>
                <f:for each="{blog.posts}" as="post">
                    <li class="post">
                        <f:render partial="PostActions" arguments="{post: post}"/>
                        <h2>
                            <f:link.action action="show" arguments="{post: post}">
↪{post.subject}</f:link.action>
                        </h2>
                        <f:render partial="PostMetaData" arguments="{post: post}"/>
                    </li>
                </f:for>
            </ul>
        </f:then>
        <f:else>
            <p>No posts created yet.</p>
        </f:else>
```

(continues on next page)

(continued from previous page)

```
    </f:if>
    <p>
        <f:link.action action="new">Create a new post</f:link.action><
    /p>
</f:section>
```

There you have it: In the first line of your template there's a reference to the "Default" 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.

The main job of this template is to display a list of the most recent posts. An `<f:if>` condition makes sure that the list of posts is only rendered if `blog` actually contains posts. But currently the view doesn't know anything about a blog - you need to adapt the the `PostController` to assign the current blog:

Classes/Acme/Blog/Controller/PostController.php:

```
/**
 * @return void
 */
public function indexAction() {
    $blog = $this->blogRepository->findActive();
    $this->view->assign('blog', $blog);
}
```

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

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

But soon you'll see that we need the current Blog in all of our actions, so how to assign it to the view without repeating the same code over and over again? With ease: We just assign it as soon as the view is initialized:

Classes/Acme/Blog/Controller/PostController.php:

```
/**
 * @param ViewInterface $view
 * @return void
 */
protected function initializeView(ViewInterface $view) {
    $blog = $this->blogRepository->findActive();
    $this->view->assign('blog', $blog);
}

/**
 * @return void
 */
public function indexAction() {
}
```

The `initializeView` method is called before each action, so it provides a good opportunity to assign values to the view that should be accessible from all actions. But make sure only to use it for truly global values in order not to waste memory for unused data.

After creating the folder `Resources/Private/Partials/` add the following two partials:

```
*Resources/Private/Partials/PostMetaData.html*:
```

```
<p class="metadata">
    Published on {post.date -> f:format.date(format: 'Y-m-d')} by {post.author}
</p>
```

Resources/Private/Partials/PostActions.html:

```
<ul class="actions">
    <li>
        <f:link.action action="edit" arguments="{post: post}">Edit</f:link.action>
    </li>
    <li>
        <f:form action="delete" arguments="{post: post}">
            <f:form.submit name="delete" value="Delete" />
        </f:form>
    </li>
</ul>
```

The `PostMetaData` partial renders date and author of a post. The `PostActions` partial an *edit* link and a button to *delete* the current post. Both are used as well in the list view (`indexAction`) as well as in the detail view (`showAction`) of the post and Partials allow us to easily re-use the parts without having to duplicate markup.

Now you should now see the list of recent posts by accessing http://dev.tutorial.local/acme.blog/post:

To create new posts and edit existing ones from the web browser, we need to create Forms:

Forms

Create a New Post

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

Classes/Acme/Blog/Controller/PostController.php:

```
/**
 * Displays the "Create Post" form
 *
 * @return void
 */
public function newAction() {
}
```

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 current `blog` is already assigned in `initializeView()` allowing the blog title and description to be rendered in our header (defined in `Default.html`). Otherwise those would be empty.

The second component is the actual form. Adjust the template `New.html` in the `Resources/Private/Templates/Post/` folder:

Resources/Private/Templates/Post/New.html:

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

(continues on next page)

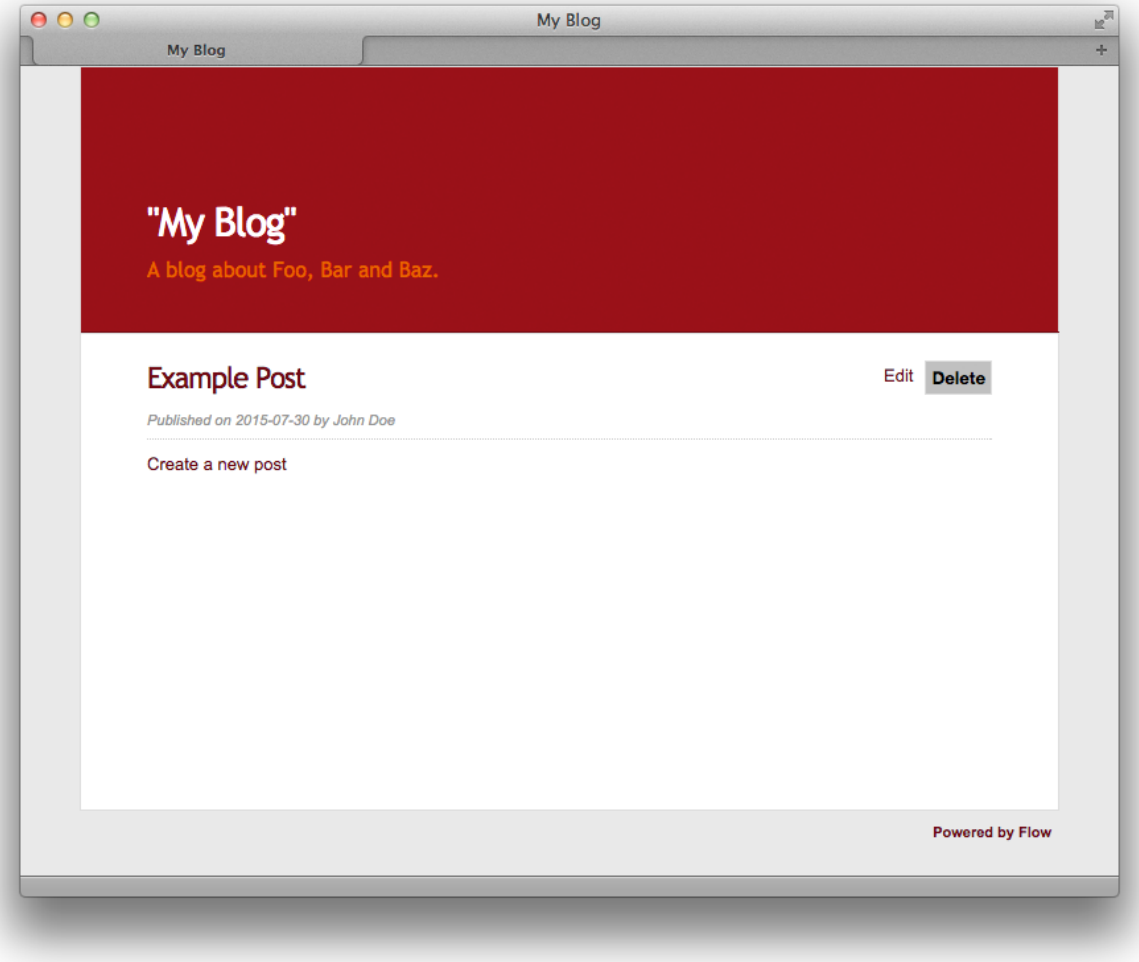

Fig. 15: The list of blog posts

(continued from previous page)

```
<f:section name="MainContent">
    <h2>Create new post</h2>
    <f:form action="create" objectName="newPost">
        <f:form.hidden property="blog" value="{blog}" />

        <label for="post-author">Author</label>
        <f:form.textfield property="author" id="post-author" />

        <label for="post-subject">Subject</label>
        <f:form.textfield property="subject" id="post-subject" />

        <label for="post-content">Content</label>
        <f:form.textarea property="content" rows="5" cols="30" id="post-content" />

        <f:form.submit name="submit" value="Publish Post" />
    </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. `objectName` finally specifies **the name of the action method argument** which will receive the form values, in this case "newPost".

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 post object. The form's elements are named after the class properties of the `Post` domain model: `blog`, `author`, `subject` and `content`. Let's look at the `createAction` again:

Note: Mind that `newPost` is not assigned to the view in this example. Assigning this object is only needed if you have set default values to your model properties. So if you for example have a `protected $hidden = TRUE` definition in your model, a `<f:form.checkbox property="hidden" />` will not be checked by default, unless you instantiate `$newPost` in your index action and assign it to the view.

Classes/Acme/Blog/Controller/PostController.php:

```
/**
 * Creates a new post
 *
 * @param Post $newPost
 * @return void
 */
public function createAction(Post $newPost) {
    $this->postRepository->add($newPost);
    $this->addFlashMessage('Created a new post.');
    $this->redirect('index');
}
```

It's important that the `createAction` uses the type hint `Post` (which expands to `\Acme\Blog\Domain\Model\Post`) and that it comes with a proper `@param` annotation because this is how Flow determines the type to which the submitted form values must be converted. Because this action requires a `Post` it gets a post (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 little plus sign above the first post lets the `newAction` render this form:

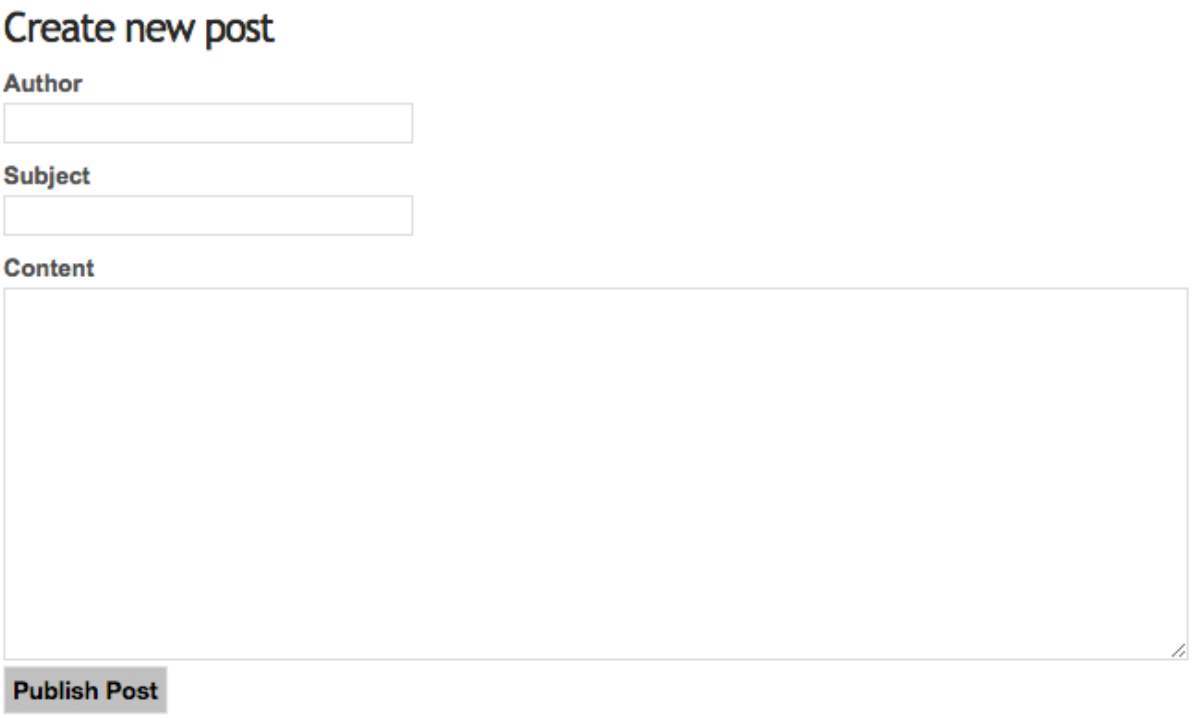

Fig. 16: Form to create a new post

Enter some data and click the submit button:

You should now find your new post in the list of posts.

Edit a Post

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

This is pretty straight forward: we already added a link to each post with the `PostActions.html` partial:

*Resources/**Private**/Templates/Post/Index.html*:

```
<ul class="actions">
    <li>
        <f:link.action action="edit" arguments="{post: post}">Edit</f:link.action>
    </li>
    <li>
        <f:form action="delete" arguments="{post: post}">
            <f:form.submit name="delete" value="Delete" />
        </f:form>
    </li>
</ul>
```

This renders an "Edit" link that points to the `editAction` of the PostController. Below is a little form with just one button that triggers the `deleteAction()`.

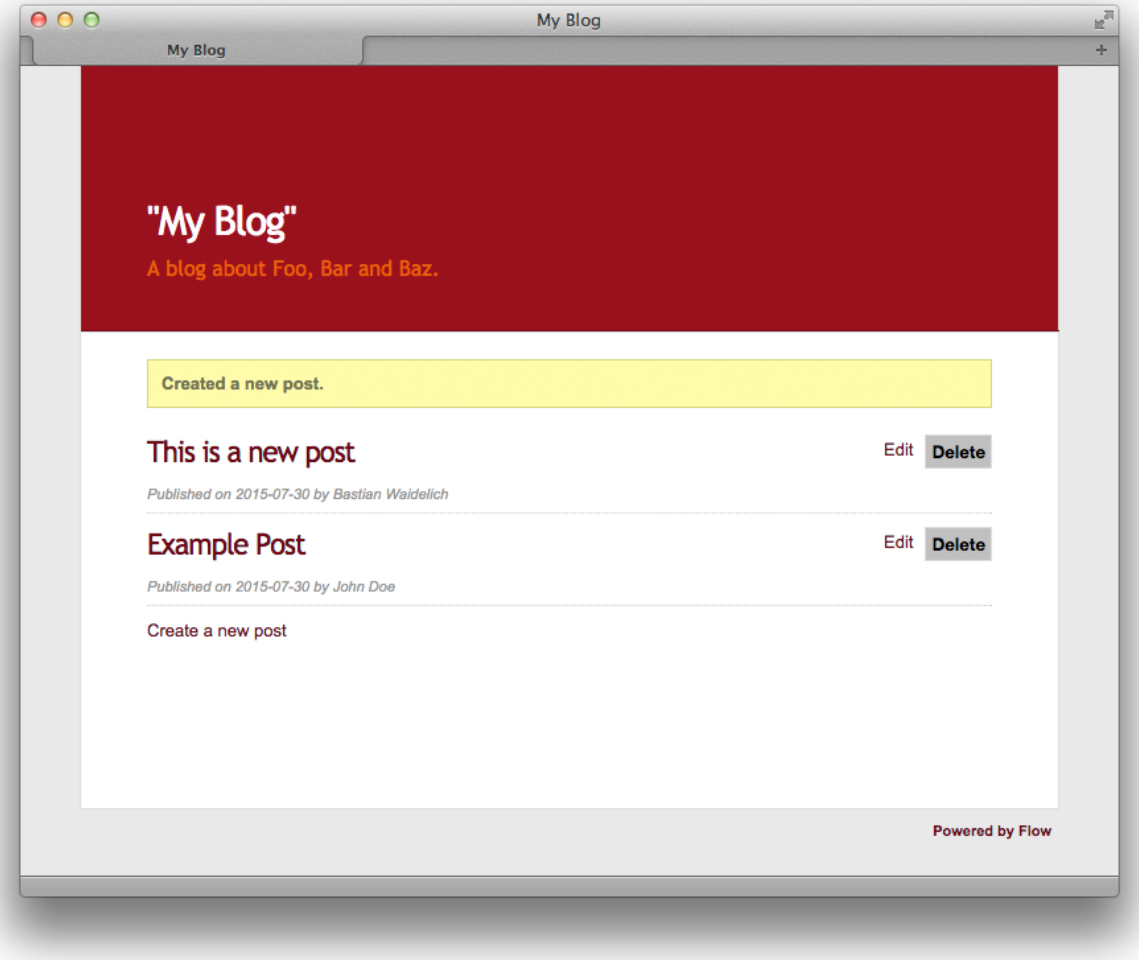

Fig. 17: A new post has been created

Note: The reason why the `deleteAction()` is invoked via a form instead of a link is because Flow follows the HTTP 1.1 specification that suggests that called "safe request methods" (usually GET or HEAD requests) should not change the server state. See *Part III - Validation* for more details. The `editAction()` just displays the Post edit form, so it can be called via GET requests.

Adjust the template `Templates/Post/Edit.html` and insert the following HTML code:

Resources/Private/Templates/Post/Edit.html:

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

<f:section name="MainContent">
    <h2>Edit post "{post.subject}"</h2>
    <f:form action="update" object="{post}" objectName="post">
        <label for="post-author">Author</label>
        <f:form.textfield property="author" id="post-author" />

        <label for="post-subject">Subject</label>
        <f:form.textfield property="subject" id="post-subject" />

        <label for="post-content">Content</label>
        <f:form.textarea property="content" rows="5" cols="30" id="post-content" />

        <f:form.submit name="submit" value="Update Post" />
    </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="{post}"` to the `<f:form>` tag. This attribute binds the variable `{post}` 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="author"` for the text box, Fluid will fetch the value of the post's `author` property and display it as the default value for the rendered text box. The resulting `input` tag will also contain the name `"author"` 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.

What's missing now is a small adjustment to the PHP code displaying the edit form:

Classes/Acme/Blog/Controller/PostController.php:

```
/**
 * Displays the "Edit Post" form
 *
 * @param Post $post
 * @return void
 */
public function editAction(Post $post) {
    $this->view->assign('post', $post);
}
```

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

When you submit the form you call the `updateAction`:

Classes/Acme/Blog/Controller/PostController.php:

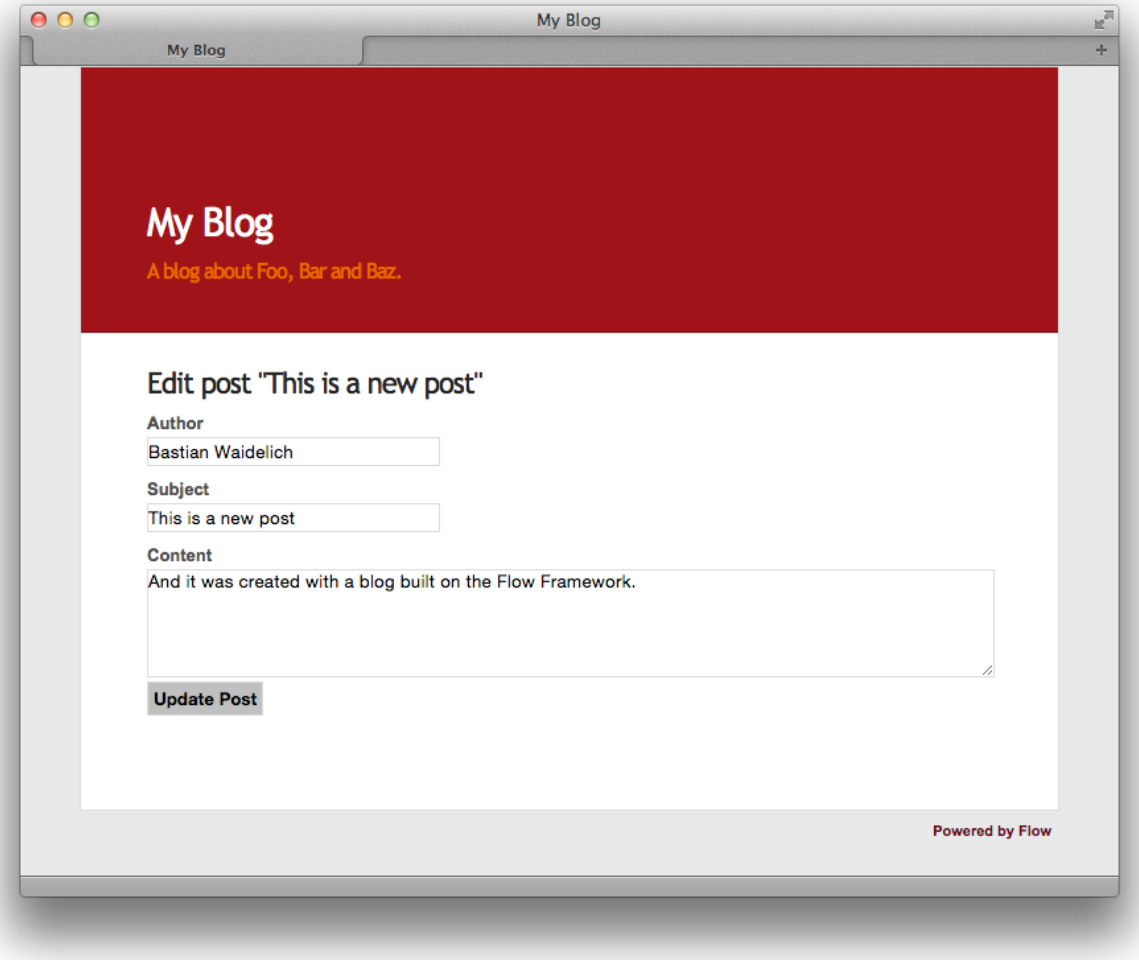

Fig. 18: The edit form for a post

```
/**
 * Updates a post
 *
 * @param Post $post
 * @return void
 */
public function updateAction(Post $post) {
    $this->postRepository->update($post);
    $this->addFlashMessage('Updated the post.');
    $this->redirect('index');
}
```

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

Fig. 19: The post has been edited

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 post**. "Edited post" means that this is a `Post` object which already contains the values submitted by the edit form.

These modifications will **not be persisted** automatically. To persist the changes to the post object, call the PostRepository's `update` method. It schedules an object for the dirty check at the end of the request.

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

```
<form action="/acme.blog/post/update" method="post">
    ...
    <input type="hidden" name="post[__identity]" value="7825fe4b-33d9-0522-a3f2-
↪02833f9084ab" />
    ...
</form>
```

Fluid automatically renders a hidden field containing information about the technical identity of the form's object, if the object is an original, previously retrieved from a repository.

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:

Table 2: 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, and set the given properties

Because the edit form contained both identity and properties, Flow prepared an instance with the given properties for our `updateAction`.

2.2.10 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 and the outgoing content before it's rendered to the browser.

You won't have to care too much about the latter if you're using Fluid to render your View because, because it *escapes* your data *by default*. As a result, even if the post subject contains the string `<script>alert("danger")</script>` outputting it via `{post.subject}` will result in the unaesthetic but harmless `&lt;script&gt;alert(&quot;danger&quot;)&lt;/script&gt;`.

But most applications come with additional rules that apply to the domain model. Maybe you want to make sure that a post subject must consist of at least 3 and at maximum 50 characters for example. But do you really want these checks in your action methods? Shouldn't we rather separate the concerns[1] of the action methods (show, create, update, ...) from others like validation, logging and security?

Fortunately Flow'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.

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. Detailed information is available in *Part III - Validation*.

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

Classes/Acme/Blog/Domain/Model/Post.php:

[1] See also: Separation of Concerns (Wikipedia)

```
/**
 * @Flow\Validate(type="NotEmpty")
 * @Flow\Validate(type="StringLength", options={ "minimum"=3, "maximum"=50 })
 * @var string
 */
protected $subject;

/**
 * @Flow\Validate(type="NotEmpty")
 * @var string
 */
protected $author;

/**
 * @Flow\Validate(type="NotEmpty")
 * @ORM\ManyToOne(inversedBy="posts")
 * @var Blog
 */
protected $blog;
```

The `Validate` annotations define one or more validation rules which should apply to a property. Multiple rules can be defined in dedicated lines by further `Validate` annotations.

Note: Per convention, every validator allows (passes) empty values, i.e. empty strings or NULL values. This is for achieving fields which are not mandatory, but if filled in, must satisfy a given validation. Consider an email address field for example which is not mandatory, but has to match an email pattern as soon as filled in.

If you want to make a field mandatory at all, use the `NotEmpty` validator in addition, like in the example above.

The technical background is the `acceptsEmptyValues` property of the AbstractValidator, being `TRUE` per default. When writing customized validators, it's basically possible to set this field to `FALSE`, however this is not generally recommended due to the convention that every validator could principally be empty.

Tip: Flow provides a range of built-in validators which can be found in the `Flow\Validation\Validator` sub package. The names used in the `type` attributes are just the unqualified class names of these validators.

It is possible and very simple to program custom validators by implementing the `Neos\Flow\Validation\Validator\ValidatorInterface`. Such validators must, however, be referred to by their fully qualified class name (i.e. including the namespace).

Make sure the above validation rules are set in your `Post` model, click on the `Create a new post` link below the list of posts and submit the *empty* form. If all went fine, you should end up again in the **new post** form, with the tiny difference that the text boxes for title and author are now framed in red:

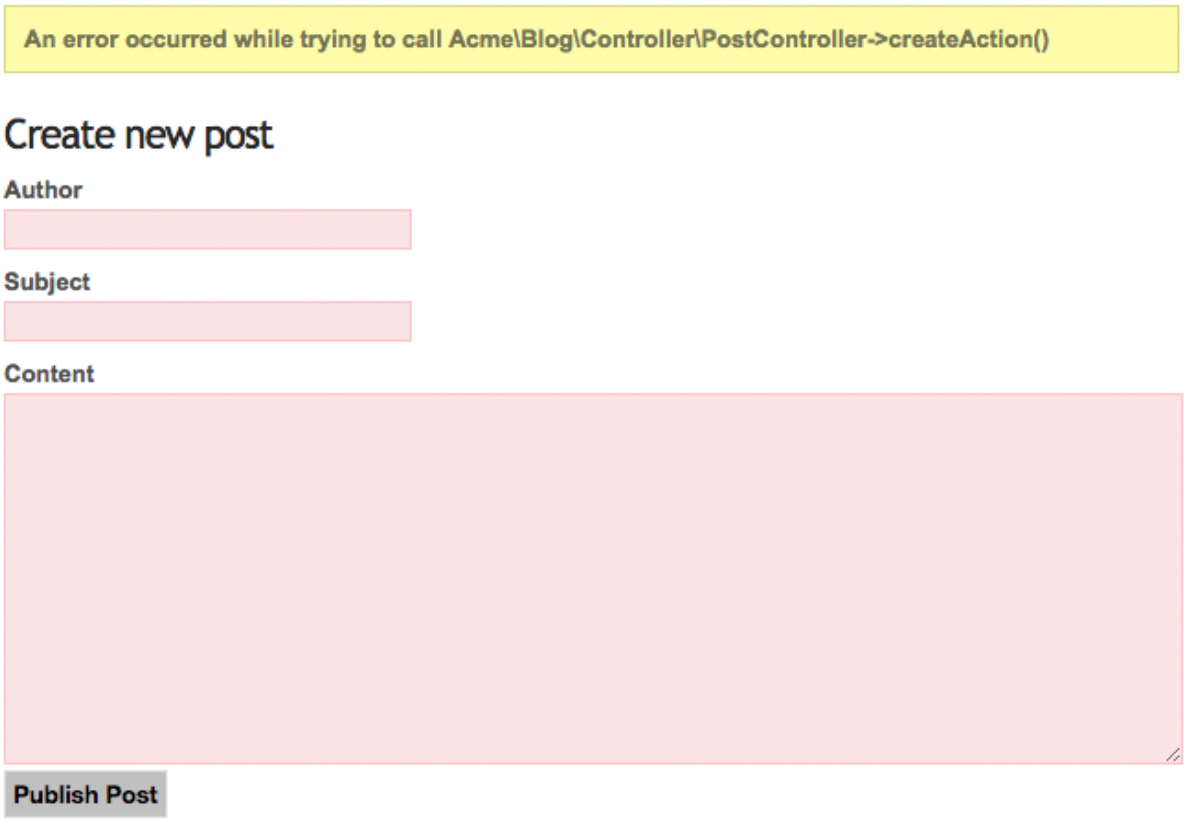

Fig. 20: Validation errors shown in form

Displaying Validation Errors

The validation rules seem to be in effect but the output could be a bit more meaningful. 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, the `<f:validation.results>` view helper. We'll need validation results for the *create* and the *update* case, so let's put the View Helper in a new partial `FormErrors`:

Resources/Private/Partials/FormErrors.html:

```
<f:validation.results for="{for}">
    <f:if condition="{validationResults.flattenedErrors}">
        <dl class="errors">
            <f:for each="{validationResults.flattenedErrors}" key="propertyName" as=
↪"errors">
                <dt>
                    {propertyName}:
                </dt>
                <dd>
                    <f:for each="{errors}" as="error">{error}</f:for>
                </dd>
            </f:for>
        </dl>
    </f:if>
</f:validation.results>
```

And include that partial to both, the `New.html` and the `Edit.html` templates just above the form:

Resources/Private/Templates/Post/New.html:

```
<f:render partial="FormErrors" arguments="{for: 'newPost'}" />
<f:form action="create" objectName="newPost">
...
```

and:

Resources/Private/Templates/Post/Edit.html:

```
<f:render partial="FormErrors" arguments="{for: 'post'}" />
<f:form action="update" object="{post}" objectName="post">
...
```

Similar to the `<f:for>` view helper `<f:validation.results>` 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 `newPost`) 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:

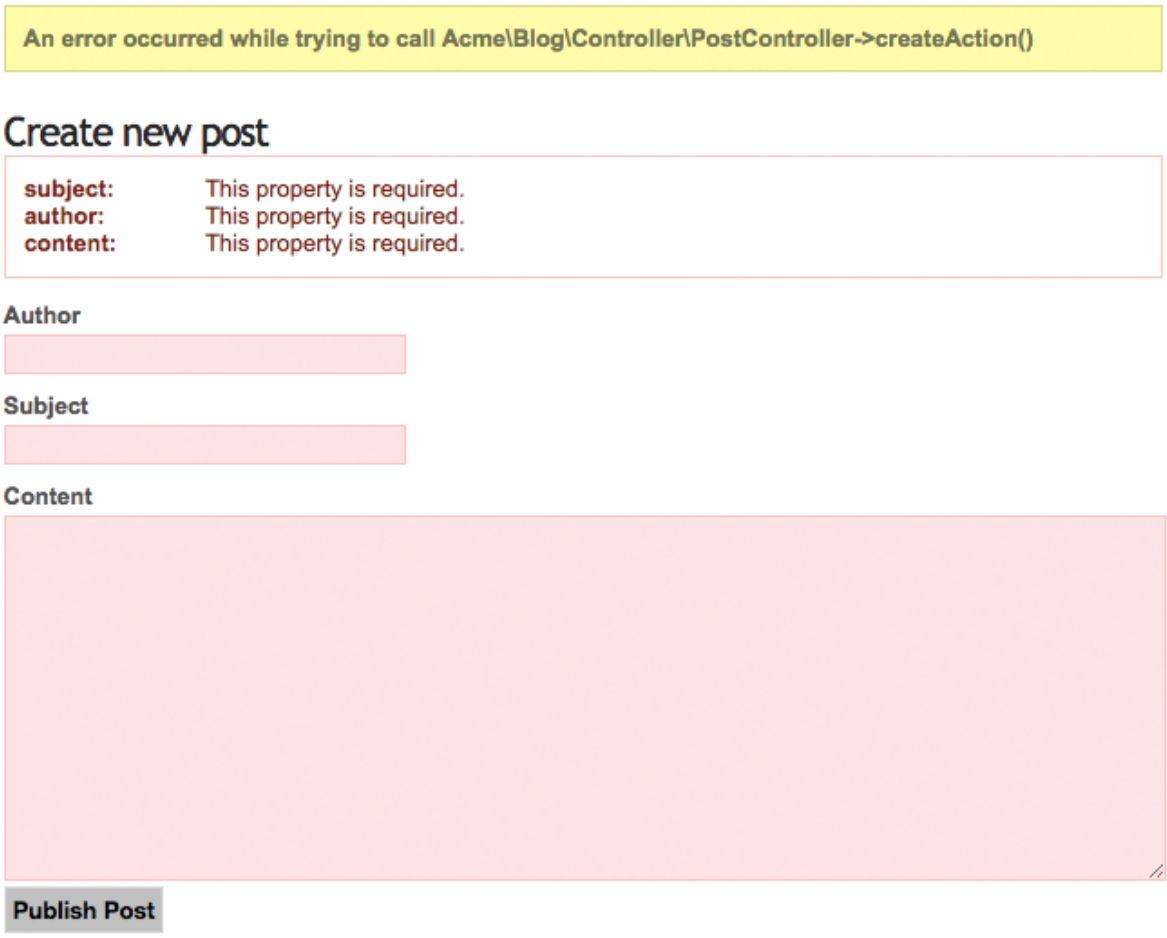

Fig. 21: More verbose validation errors shown in form

Validating Existing Data

The validation rules are enforced as soon as the GET or POST arguments are mapped to the action's arguments. But what if you add new validation rules when there are already persisted entities that might violate these? For example if you had created a post with a subject of "xy" and added the `StringLength` annotation afterwards?

Doing so would prevent you from invoking any of the actions for that particular post. All you will see is an error message:

```
Validation failed while trying to call Acme\Blog\Controller\PostController->
↪showAction().
```

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

There's a very simple remedy to this problem: don't validate the post. With one additional annotation the whole mechanism works as expected:

Classes/Acme/Blog/Controller/PostController.php:

```
/**
 * Displays a single post
 *
 * @Flow\IgnoreValidation("$post")
 * @param Post $post
 * @return void
 */
public function showAction(Post $post) {
    $this->view->assignMultiple([
        'post' => $post,
        'nextPost' => $this->postRepository->findNext($post),
        'previousPost' => $this->postRepository->findPrevious($post),
    ]);
}
```

Now the `showAction` can be called even though `$post` is not valid. You probably want to add the same annotation to the `editAction` and even the `deleteAction` so that invalid posts can be fixed or removed.

2.2.11 Routing

Although the basic functions like creating or updating a post work well already, the URIs still have a little blemish. The index of posts can only be reached by the cumbersome address http://dev.tutorial.local/acme.blog/post and the URL for editing a post refers to the post's UUID instead of the human-readable identifier.

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

Post Index Route

Our first task is to simplify accessing the list of posts. For that you need to edit a file called *Routes.yaml* in the global *Configuration/* 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 'Welcome' 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: 'Post index'
  uriPattern: 'posts'
  defaults:
    '@package':    'Acme.Blog'
    '@controller': 'Post'
    '@action':     'index'
    '@format':     'html'
```

This configuration adds a new route to the list of routes (- creates a new list item). The route becomes active if a requests matches the pattern defined by the `uriPattern`. In this example the URI http://dev.tutorial.local/posts would match.

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/posts now – you should see the list of posts produced by the `PostController`'s `indexAction`.

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 *Flow* sub route configuration for example includes further routes if no earlier routes in `Routes.yaml` matched first. Only the URI part contained in the less-than and greater-than signs will be passed to the sub routes:

```
##
# Flow subroutes
#

-
  name: 'Flow'
  uriPattern: '<FlowSubroutes>'
  defaults:
    '@format': 'html'
  subRoutes:
    'FlowSubroutes':
      package: 'Neos.Flow'
```

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

-
```

(continues on next page)

(continued from previous page)

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

Note: We use "`BlogSubroutes`" here as name for the sub routes. You can name this as you like but it has to be the same in `uriPattern` and `subRoutes`.

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

Configuration/Routes.yaml:

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

-
  name: 'Post index'
  uriPattern: 'posts'
  defaults:
    '@package':    'Acme.Blog'
    '@controller': 'Post'
    '@action':     'index'
    '@format':     'html'
```

Note: As the defaults for `@package` and `@format` are already defined in the parent route, you can omit them in the sub route.

An Action Route

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

Configuration/Routes.yaml:

```
-
  name: 'Post actions'
  uriPattern: 'posts/{@action}'
  defaults:
    '@controller': 'Post'
```

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

However, the edit link still looks it bit ugly:

```
http://dev.tutorial.local/acme.blog/post/edit?post%5B__identity%5D=229e2b23-b6f3-4422-
↪8b7a-efb196dbc88b
```

Fig. 22: A nice "create" route

For getting rid of this long identifier we need the help of a new route that can handle the post object.

Object Route Parts

Our goal is to produce an URI like:

```
http://dev.tutorial.local/posts/2010/01/18/post-title/edit
```

and use this as our edit link. That's done by adding following route at the **top of the file**:

Configuration/Routes.yaml:

```
-
  name: 'Single post actions'
  uriPattern: 'posts/{post}/{@action}'
  defaults:
    '@controller':  'Post'
  routeParts:
    post:
      objectType: 'Acme\Blog\Domain\Model\Post'
      uriPattern: '{date:Y}/{date:m}/{date:d}/{subject}'
```

The "`Single post actions`" route now handles all actions where a post needs to be specified (i.e. show, edit, update and delete).

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

More on Routing

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

More information on routing can be found in the *The Definitive Guide*.

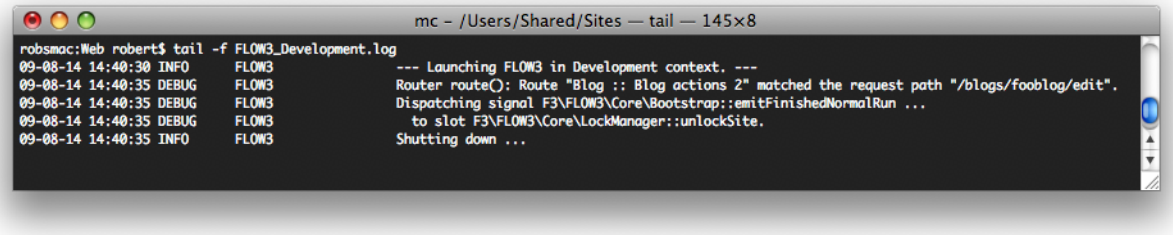

Fig. 23: Routing entries in the system log

2.2.12 Summary

Next Steps

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

You now have two options for delving further into Flow programming:

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

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

- Delete your blog package (that is *Packages/Application/ACME.Blog/*) and then
- Clone the Blog package from github: https://github.com/neos/Acme.Blog

Feedback

The Flow core team is curious about getting your feedback! If you have any questions, are stuck at some point or just want to let us know how you liked the tutorial please join us at Slack or open a thread on our forum.

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

2.3 Part III: Manual

2.3.1 Architectural Overview

Flow 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 Flow.

Although we created Flow as the foundation for the Neos Content Management System, its approach is general enough to be useful as a basis for any other PHP application. We're happy to share the Flow framework with the whole PHP community and are looking forward to the hundreds of new features and enhancements contributed as packages by other enthusiastic developers.

This reference describes all features of Flow and provides you with in-depth information. If you'd like to get a feeling for Flow and get started quickly, we suggest that you try out our Getting Started tutorial first.

System Parts

The Flow framework is composed of the following submodules:

- The *Flow Bootstrap* takes care of configuring and initializing the whole framework.
- The *Package* Manager allows you to download, install, configure and uninstall packages.
- The *ObjectManagement* is in charge of building, caching and combining objects.
- The *Configuration* framework reads and cascades various kinds of configuration from different sources and provides access to it.
- The *ResourceManagement* module contains functions for publishing, caching, securing and retrieving resources.
- The *HTTP* component is a standards-compliant implementation of a number of RFCs around HTTP, Cookies, content negotiation and more. It is based on the PHP-FIG PSR-15 and PSR-7 specifications.
- The *MVC* framework takes care of requests and responses and provides you with a powerful, easy-to use Model-View-Controller implementation.
- The *Cli* module provides a very easy way to implement CLI commands using Flow, including built-in help based on code documentation.
- The *Cache* framework provides different kinds of caches with can be combined with a selection of cache backends.
- The *Error* module handles errors and exceptions and provides utility classes for this purpose.
- The *Log* module provides simple but powerful means to log any kind of event or signal into different types of backends.
- The *Signal Slot* module implements the event-driven concept of signals and slots through AOP aspects.
- The *Validation* module provides a validation and filtering framework with built-in rules as well as support for custom validation of any object.
- The *Property* module implements the concept of property editors and is used for setting and retrieving object properties.
- The *Reflection* API complements PHP's built-in reflection support by advanced annotation handling and a cached reflection service.
- The *AOP* framework enables you to use the powerful techniques of Aspect Oriented Programming.
- The *Persistence* module allows you to transparently persist your objects following principles of *Domain Driven Design*.
- The *Security* framework enforces security policies and provides an API for managing those.
- The *Session* framework takes care of session handling and storing session information in different backends
- The *I18n* service manages languages and other regional settings and makes them accessible to other packages and Flow sub packages.
- The *Utility* module is a library of useful general-purpose functions for file handling, algorithms, environment abstraction and more.

If you are overwhelmed by the amount of information in this reference, just keep in mind that you don't need to know all of it to write your own Flow packages. You can always come back and look up a specific topic once you need to know about it - that's what references are for.

But even if you don't need to know everything, we recommend that you get familiar with the concepts of each module and read the whole manual. This way you make sure that you don't miss any of the great features Flow provides and hopefully feel inspired to produce clean and easy-maintainable code.

2.3.2 Bootstrapping

This chapter outlines the bootstrapping mechanism Flow uses on each request to initialize vital parts of the framework and the application. It explains the built-in request handlers which effectively control the boot sequence and demonstrates how custom request handlers can be developed and registered.

The Flow Application Context

Each request, no matter if it runs from the command line or through HTTP, runs in a specific *application context*. Flow provides exactly three built-in contexts:

- `Development` (default) - used for development
- `Production` - should be used for a live site
- `Testing` - is used for functional tests

The context Flow runs in is specified through the environment variable `FLOW_CONTEXT`. It can be set per command at the command line or be part of the web server configuration:

```
# run the Flow CLI commands in production context
FLOW_CONTEXT=Production ./flow help

# In your Apache configuration, you usually use:
SetEnv FLOW_CONTEXT Production
```

Custom Contexts

In certain situations, more specific contexts are desirable:

- a staging system may run in a Production context, but requires a different set of credentials than the production server.
- developers working on a project may need different application specific settings but prefer to maintain all configuration files in a common Git repository.

By defining custom contexts which inherit from one of the three base contexts, more specific configuration sets can be realized.

While it is not possible to add new "top-level" contexts at the same level like *Production* and *Testing*, you can create arbitrary *sub-contexts*, just by specifying them like `<MainContext>/<SubContext>`.

For a staging environment a custom context `Production/Staging` may provide the necessary settings while the `Production/Live` context is used on the live instance.

Each sub context inherits the configuration from the parent context, which is explained in full detail inside the *Configuration* chapter.

Note: This even works recursively, so if you have a multiple-server staging setup, you could use the context Production/Staging/Server1 and Production/Staging/Server2 if both staging servers needed different configuration.

Boot Sequence

There are basically two types of requests which are handled by a Flow application:

- *command line* requests are passed to the `flow.php` script which resides in the `Scripts` folder of the Flow package
- *HTTP requests* are first taken care of by the `index.php` script in the public `Web` directory.

Both scripts set certain environment variables and then instantiate and run the `Neos\Flow\Core\Bootstrap` class.

The bootstrap's `run()` method initializes the bare minimum needed for any kind of operation. When it did that, it determines the actual request handler which takes over the control of the further boot sequence and handling the request.

```
public function run() {
        Scripts::initializeClassLoader($this);
        Scripts::initializeSignalSlot($this);
        Scripts::initializePackageManagement($this);

        $this->activeRequestHandler = $this->resolveRequestHandler();
        $this->activeRequestHandler->handleRequest();
}
```

The request handler in charge executes a sequence of steps which need to be taken for initializing Flow for the purpose defined by the specialized request handler. Flow's `Bootstrap` class provides convenience methods for building such a sequence and the result can be customized by adding further or removing unnecessary steps.

After initialization, the request handler takes the necessary steps to handle the request, does or does not echo a response and finally exits the application. Control is not returned to the bootstrap again, but a request handler should call the bootstrap's `shutdown()` method in order to cleanly shut down important parts of the framework.

Run Levels

There are two pre-defined levels to which Flow can be initialized:

- *compiletime* brings Flow into a state which allows for code generation and other low-level tasks which can only be done while Flow is not yet fully ready for serving user requests. Compile time has only limited support for Dependency Injection and lacks support for many other functions such as Aspect-Oriented Programming and Security.
- *runtime* brings Flow into a state which is fully capable of handling user requests and is optimized for speed. No changes to any of the code caches or configuration related to code is allowed during runtime.

The bootstrap's methods `buildCompiletimeSequence()` and `buildRuntimeSequence()` conveniently build a sequence which brings Flow into either state on invocation.

Request Handlers

A request handler is in charge of executing the boot sequence and ultimately answering the request it was designed for. It must implement the `\Neos\Flow\Core\RequestHandlerInterface` interface which, among others, contains the following methods:

```
public function handleRequest();

public function canHandleRequest();

public function getPriority();
```

On trying to find a suitable request handler, the bootstrap asks each registered request handler if it can handle the current request using `canHandleRequest()` – and if it can, how eager it is to do so through `getPriority()`. It then passes control to the request handler which is most capable of responding to the request by calling `handleRequest()`.

Request handlers must first be registered in order to be considered during the resolving phase. Registration is done in the `Package` class of the package containing the request handler:

```
class Package extends BasePackage {

        public function boot(\Neos\Flow\Core\Bootstrap $bootstrap) {
                $bootstrap->registerRequestHandler(new \Acme\Foo\BarRequestHandler(
↪$bootstrap));
        }

}
```

Tip: The Flow package contains meaningful working examples for registration of request handlers and building boot sequences. A good starting point is the `\Neos\Flow\Package` class where the request handlers are registered.

2.3.3 Package Management

Flow is a package-based system. In fact, Flow itself is just a package as well - but obviously an important one. Packages act as a container for different matters: Most of them contain PHP code which adds certain functionality, others only contain documentation and yet other packages consist of templates, images or other resources.

Package Locations

Framework and Application Packages

Flow packages are located in a sub folder of the *Packages/* directory. A typical application (such as Neos for example) will use the core packages which are bundled with Flow and use additional packages which are specific to the application. The framework packages are kept in a directory called *Framework* while the application specific packages reside in the *Application* directory. This leads to the following folder structure:

Configuration/ The global configuration folder

Data/ The various data folders, temporary as well as persistent

Packages/

> **Framework/** The Framework directory contains packages of the Flow distribution.

Application/ The Application directory contains your own / application specific packages.

Libraries/ The Libraries directory contains 3rd party packages.

Additional Package Locations

Apart from the *Application*, *Framework* and *Libraries* package directories you may define your very own additional package locations by just creating another directory in the application's *Packages* directory. One example for this is the Neos distribution, which expects packages with website resources in a folder named *Sites*.

The location for Flow packages installed via Composer (as opposed to manually placing them in a *Packages/* sub folder) is determined by looking at the package type in the manifest file. This would place a package into *Packages/Acme*:

```
"type": "neos-acme"
```

If you would like to use `package:create` to create packages of this type in *Packages/Acme* instead of the default location *Packages/Application*, add an entry in the *Settings.yaml* of the package that expects packages of that type:

```
Neos:
  Flow:
    package:
      packagesPathByType:
        'neos-acme': 'Acme'
```

Note: Packages where the type starts with `typo3-flow-` or `neos-` are considered Flow packages and will therefore be reflected and proxied by default. We recommend using only the `neos-` prefix for the type when creating new packages (but only from Flow 3.2 upwards) as the other is deprecated and will stop working in the next major.

Package Directory Layout

The Flow package directory structure follows a certain convention which has the advantage that you don't need to care about any package-related configuration. If you put your files into the right directories, everything will just work.

The directory layout inside a Flow package is as follows:

Classes This directory contains the actual source code for the package. Package authors are free to add (only!) class or interface files directly to this directory or add subdirectories to organize the content as necessary. All classes or interfaces below this directory are handled by the autoloading mechanism and will be registered at the object manager automatically (and will thus be considered "registered objects").

One special file in here is the *Package.php* which contains the class with the package's bootstrap code (if needed).

Configuration All kinds of configuration which are delivered with the package reside in this directory. The configuration files are immutable and must not be changed by the user or administrator. The most prominent configuration files are the *Objects.yaml* file which may be used to configure the package's objects and the *Settings.yaml* file which contains general user-level settings.

Documentation Holds the package documentation. Please refer to the Documenter's Guide for more details about the directories and files within this directory.

Resources Contains static resources the package needs, such as library code, template files, graphics, ... In general, there is a distinction between public and private resources.

Private Contains private resources for the package. All files inside this directory will never be directly available from the web.

Installer/Distribution The files in this directory are copied to the root of a Flow installation when the package is installed or updated via Composer. Anything in `Defaults` is copied only, if the target does not exist (files are not overwritten). Files in `Essentials` are overwritten and thus kept up-to-date with the package they come from.

Templates Template files used by the package should go here. If a user wants to modify the template it will end up elsewhere and should be pointed to by some configuration setting.

PHP Should hold any PHP code that is an external library which should not be handled by the object manager (at least not by default), is of procedural nature or doesn't belong into the classes directory for any other reason.

Java Should hold any Java code needed by the package. Repeat and rinse for Smalltalk, Modula, Pascal, ... ;)

Public Contains public resources for the package. All files in this directory will be mirrored into Flow's *Web* directory by the ResourceManager (and therefore become accessible from the web). They will be delivered to the client directly without further processing.

Although it is up to the package author to name the directories, we suggest the following directories:

- Images
- Styles
- Scripts

The general rule for this is: The folder uses the plural form of the resource type it contains.

Third party bundles that contain multiple resources such as `jQuery UI` or `Twitter Bootstrap` should reside in a sub directory `Libraries`.

Tests

Unit Holds the unit tests for the package.

Functional Holds the functional tests for the package.

As already mentioned, all classes which are found in the *Classes* directory will be detected and registered. However, this only works if you follow the naming rules equally for the class name as well as the filename. An example for a valid class name is `\MyCompany\MyPackage\Controller\StandardController` while the file containing this class would be named *StandardController.php* and is expected to be in a directory *MyCompany.MyPackage/Classes/MyCompany/MyPackage/Controller*.

All details about naming files, classes, methods and variables correctly can be found in the Flow Coding Guidelines. You're highly encouraged to read (and follow) them.

Package Keys

Package keys are used to uniquely identify packages and provide them with a namespace for different purposes. They save you from conflicts between packages which were provided by different parties.

We use *vendor namespaces* for package keys, i.e. all packages which are released and maintained by the Neos and Flow core teams start with `Neos.*` (for historical reasons) or `Neos.*`. In your company we suggest that you use your company name as vendor namespace.

To define the package key for your package we recommend you set the "extra.neos.package-key" option in your composer.json as in the following example:

composer.json:

```
"extra": {
    "neos": {
        "package-key": "Vendor.PackageKey"
    }
}
```

Loading Order

The loading order of packages follows the dependency chain as defined in the composer manifests involved, solely taking the "require" part into consideration. Additionally you can configure packages that should be loaded before by adding an array of composer package names to "extra.neos.loading-order.after" as in this example:

composer.json:

```
"extra": {
    "neos": {
        "loading-order": {
            "after": [
                "some/package"
            ]
        }
    }
}
```

Installing a Package

There are various ways of installing packages. They can just be copied to a folder in *Packages/*, either manually or by some tool, or by keeping them in your project's VCS tool (directly or indirectly, via git submodules or svn:externals).

The true power of dependency management comes with the use of Composer, though. Installing a package through composer allows to install dependencies of that package automatically as well. That is why we suggest only using composer to install packages.

If a package you would like to add is available on Packagist it can be installed by running:

```
composer require <vendor/package>
```

Note: If you need to install Composer first, read the installation instructions

In case a package is not available through Packagist, you can still install via Composer as it supports direct fetching from popular SCM system. For this, define a repository entry in your manifest to be able to use the package name as usual in the dependencies.

composer.json:

```
"repositories": [
    {
        "type": "git",
        "url": "git://github.com/acme/demo.git"
    },
    ...
],
...
```

(continues on next page)

(continued from previous page)

```
"require": {
    ...,
    "acme/demo": "dev-master"
}
```

Creating a New Package

Use the `package:create` command to create a new package:

```
$ ./flow package:create Acme.Demo
```

This will create the package in *Packages/Application*. After that, adjust *composer.json* to your needs. Apart from that no further steps are necessary.

Updating Packages

The packages installed via Composer can be updated with the command:

```
composer update
```

Package Meta Information

All packages need to provide some meta information to Flow. The data is split in two files, depending on primary use.

composer.json

The Composer manifest. It declares metadata like the name of a package as well as dependencies, like needed PHP extensions, version constraints and other packages. For details on the format and possibilities of that file, have a look at the Composer documentation.

Classes/Package.php

This file contains bootstrap code for the package. If no bootstrap code is needed, it does not need to exist.

Example: Minimal Package.php

```
<?php
namespace Acme\Demo;

use Neos\Flow\Package\Package as BasePackage;

/**
 * The Acme.Demo Package
 *
 */
class Package extends BasePackage {

        /**
        * Invokes custom PHP code directly after the package manager has been␣
↪initialized.
```

(continues on next page)

(continued from previous page)

```
        *
        * @param \Neos\Flow\Core\Bootstrap $bootstrap The current bootstrap
        * @return void
        */
        public function boot(\Neos\Flow\Core\Bootstrap $bootstrap) {
                $bootstrap->registerRequestHandler(new \Acme\Demo\Quux\RequestHandler(
↪$bootstrap));

                $dispatcher = $bootstrap->getSignalSlotDispatcher();
                $dispatcher->connect(\Neos\Flow\Mvc\Dispatcher::class,
↪'afterControllerInvocation', \Acme\Demo\Baz::class, 'fooBar');
        }
}
?>
```

The bootstrap code can be used to wire some signal to a slot or to register request handlers (as shown above), or anything else that can must be done early the bootstrap stage.

After creating a new `Package.php` in your package you need to execute:

```
$ ./flow flow:package:rescan
```

Otherwise the `Package.php` will not be found.

Using Third Party Packages

When using 3rd party packages via Composer everything should work as expected. Flow uses the Composer autoloader to load code. Third party packages will not have any Flow "magic" enabled by default. That means no AOP will work on classes from third party packages. If you need this see *Enabling Other Package Classes For Object Management*

2.3.4 Configuration

Configuration is an important aspect of versatile applications. Flow provides you with configuration mechanisms which have a small footprint and are convenient to use and powerful at the same time. Hub for all configuration is the configuration manager which handles alls configuration tasks like reading configuration, configuration cascading, and (later) also writing configuration.

File Locations

There are several locations where configuration files may be placed. All of them are scanned by the configuration manager during initialization and cascaded into a single configuration tree. The following locations exist (listed in the order they are loaded, i.e. later values override prior ones):

`/Packages/<PackageDirectoryAndName>/Configuration/` The *Configuration* directory of each package is scanned first. Only at this stage new configuration options must be introduced (by defining a default value).

`/Configuration/` Configuration in the global *Configuration* directory overrides the default settings defined in the package's configuration directories.

`/Packages/<PackageDirectoryAndName>/Configuration/<ApplicationContext>/` There may exist a subdirectory for each application context (see Flow Bootstrap section). This configuration is only loaded if Flow runs in the respective application context.

/Configuration/<ApplicationContext>/ The context specific configuration again overrides the generic settings.

The configuration manager also considers custom contexts, such as `Production/Live`. First, the base configuration is loaded, followed by the context specific configuration for `Production` and `Production/Live`.

Flow's configuration system does not support placing configuration files anywhere except for in `Configuration/` or one of the context directories in `Configuration/`. Flow only supports three top-level contexts: Production, Development, and Testing. These folders are reserved for the Flow configuration system.

Configuration Files

Flow distinguishes between different types of configuration. The most important type of configuration are the settings, however other configuration types exist for special purposes.

The configuration format is YAML and the configuration options of each type are defined in their own dedicated file:

Settings.yaml Contains user-level settings, i.e. configuration options the users or administrators are meant to change. Settings are the highest level of system configuration. Settings have *Split configuration sources* enabled.

Routes.yaml Contains routes configuration. This routing information is parsed and used by the MVC Web Routing mechanism. Refer to the *Routing* chapter for more information.

Objects.yaml Contains object configuration, i.e. options which configure objects and the combination of those on a lower level. See the *Object Framework* chapter for more information. Objects have *Split configuration sources* enabled.

Policy.yaml Contains the configuration of the security policies of the system. See the *Security* chapter for details. Policy has *Split configuration sources* enabled.

PackageStates.php Contains a list of packages and their current state, for example if they are active or not. Don't edit this file directly, rather use the *flow* command line tool do activate and deactivate packages.

Caches.yaml Contains a list of caches which are registered automatically. Caches defined in this configuration file are registered in an early stage of the boot process and profit from mechanisms such as automatic flushing by the File Monitor. See the chapter about the *Cache Framework* for details. Caches have *Split configuration sources* enabled.

Views.yaml Contains configurations for Views, for example the lookup paths for templates. See the *Model View Controller* chapter for details.

Defining Configuration

Configuration Format

The format of Flow's configuration files is YAML. YAML is a well-readable format which is especially well-suited for defining configuration. The full specification among with many examples can be found on the YAML website. All important parts of the YAML specification are supported by the parser used by Flow, it might happen though that some exotic features won't have the desired effect. At best you look at the configuration files which come with the Flow distribution for getting more examples.

Example: a package-level Settings.yaml

```
#                                                                          #
# Settings Configuration for the Neos.Viewhelpertest Package               #
#                                                                          #
```

(continues on next page)

(continued from previous page)

```
Neos:
  Viewhelpertest:
    includeViewHelpers: [alias, base]

    xhprof:
      rootDirectory: '' # path to the XHProf library
      outputDirectory: '%FLOW_PATH_DATA%Temporary/Viewhelpertest/XHProf/' # output␣
↪directory

    profilingTemplatesDirectory: '%FLOW_PATH_DATA%Temporary/Viewhelpertest/
↪Fluidtemplates/'
```

Warning: Always use *two spaces* for indentation in YAML files. The parser will not accept indentation using tabs.

Constants and Environment

Sometimes it is necessary to use values in your configuration files which are defined as PHP constants or are environment variables. These values can be included by special markers which are replaced by the actual value during parse time. The format is `%<CONSTANT_NAME>%` where `<CONSTANT_NAME>` is the name of a constant or `%env:<ENVIRONMENT_VARIABLE>%`. Note that the constant or environment variable name must be all uppercase.

Some examples:

`%FLOW_PATH_WEB%` Will be replaced by the path to the public web directory.

`%FLOW_PATH_DATA%` Will be replaced by the path to the */Data/* directory.

`%PHP_VERSION%` Will be replaced by the current PHP version.

`%Neos\Flow\Core\Bootstrap::MINIMUM_PHP_VERSION%` Will be replaced by this class constant's value. Note that a leading namespace backslash is generally allowed as of PHP, but is not recommended due to CGL (stringed class names should not have a leading backslash).

`%env:HOME%` Will be replaced by the value of the "HOME" environment variable.

Custom Configuration Types

Custom configuration types allow to extract parts of the system configuration into separate files.

The following will register a new type `Views` for configuration, using the default configuration processing handler. The code needs to be in your `Package``s ``boot()` method.

Example: Register a custom configuration type

```
$dispatcher = $bootstrap->getSignalSlotDispatcher();
$dispatcher->connect(\Neos\Flow\Configuration\ConfigurationManager::class,
↪'configurationManagerReady',
    function ($configurationManager) {
        $configurationManager->registerConfigurationType('Views');
    }
);
```

This will allow to use the new configuration type `Views` in the same way as the other types supported by Flow natively, as soon as you have a file named `Views.yaml` in your configuration folder(s). See *Working with other configuration* for details.

If you want to use a specific configuration processing type, you can pass it when registering the configuration. The supported types are defined as `CONFIGURATION_PROCESSING_TYPE_*` constants in `ConfigurationManager`.

Example: Register a custom configuration type

```
$dispatcher = $bootstrap->getSignalSlotDispatcher();
$dispatcher->connect(\Neos\Flow\Configuration\ConfigurationManager::class,
↪'configurationManagerReady',
    function ($configurationManager) {
        $configurationManager->registerConfigurationType(
            'CustomObjects',
            ConfigurationManager::CONFIGURATION_PROCESSING_TYPE_OBJECTS
        );
    }
);
```

Split configuration sources

For custom types it is possible to allow for *split* configuration sources. For the YAML source used in Flow it allows to use the configuration type as a prefix for the configuration filenames.

Example: Register a custom configuration type, split-source

```
$dispatcher = $bootstrap->getSignalSlotDispatcher();
$dispatcher->connect(\Neos\Flow\Configuration\ConfigurationManager::class,
↪'configurationManagerReady',
    function (ConfigurationManager $configurationManager) {
        $configurationManager->registerConfigurationType(
            'Models',
            ConfigurationManager::CONFIGURATION_PROCESSING_TYPE_DEFAULT,
            true
        );
    }
);
```

The above code will lead to the following files being read, sorted by name and merged if the configuration of type `Models` is requested:

```
Configuration/
    Models.yaml
    Models.Foo.yaml
    Models.Bar.yaml
    Models.Quux.yaml
```

Note: Split configuration is supported for all except `CONFIGURATION_PROCESSING_TYPE_ROUTES` processing types. This is because Routing uses a custom include semantic that shares the naming convention with split sources.

Accessing Settings

In almost all cases, Flow will automatically provide you with the right configuration.

What you usually want to work with are `settings`, which are application-specific to your package. The following example demonstrates how to let Flow inject the settings of a classes' package and output some option value:

Example: Settings Injection

```yaml
Acme:
  Demo:
    administrator:
      email: 'john@doe.com'
      name: 'John Doe'
```

```php
namespace Acme\Demo;

class SomeClass {

    /**
     * @var array
     */
    protected $settings;

    /**
     * Inject the settings
     *
     * @param array $settings
     * @return void
     */
    public function injectSettings(array $settings) {
        $this->settings = $settings;
    }

    /**
     * Outputs some settings of the "Demo" package.
     *
     * @return void
     */
    public function theMethod() {
        echo ($this->settings['administrator']['name']);
        echo ($this->settings['administrator']['email']);
    }
}
```

Note: Injecting all settings creates tight coupling to the settings. If you only need a few settings you might want to inject those specifically with the Inject annotation described below.

Injection of single settings into properties

Flow provides a way to inject specific settings through the `InjectConfiguration` annotation directly into your properties. The annotation provides three optional attributes related to configuration injection:

- `package` specifies the package to get the configuration from. Defaults to the package the current class belongs to.
- `path` specifies the path to the setting that should be injected. If it's not set all settings of the current (or
- `type` one of the ConfigurationManager::CONFIGURATION_TYPE_* constants to define where the configuration is fetched from, defaults to ConfigurationManager::CONFIGURATION_TYPE_SETTINGS.

Note: As a best-practice for testing and extensibility you should also provide setters for any setting you add to your class, although this is not required for the injection to work.

Example: single setting injection

```yaml
Acme:
  Demo:
    administrator:
      name: 'John Doe'
SomeOther:
  Package:
    email: 'john@doe.com'
```

```php
namespace Acme\Demo;

use Neos\Flow\Annotations as Flow;

class SomeClass {

  /**
   * @Flow\InjectConfiguration(path="administrator.name")
   * @var string
   */
  protected $name;

  /**
   * @Flow\InjectConfiguration(package="SomeOther.Package", path="email")
   * @var string
   */
  protected $email;

  /**
   * @Flow\InjectConfiguration(package="SomeOther.Package")
   * @var array
   */
  protected $someOtherPackageSettings = array();

  /**
   * Overrides the name
   *
   * @param string $name
   * @return void
   */
```

(continues on next page)

(continued from previous page)

```
  public function setName($name) {
    $this->name = $name;
  }

  /**
   * Overrides the email
   *
   * @param string $email
   * @return void
   */
  public function setEmail($email) {
    $this->email = $email;
  }
}
```

Working with other configuration

Although infrequently necessary, it is also possible to retrieve options of the more special configuration types. The `ConfigurationManager` provides a method called `getConfiguration()` for this purpose. The result this method returns depends on the actual configuration type you are requesting.

Bottom line is that you should be highly aware of what you're doing when working with these special options and that they might change in a later version of Flow. Usually there are much better ways to get the desired information (e.g. ask the Object Manager for object configuration).

Configuration Cache

Parsing the YAML configuration files takes a bit of time which remarkably slows down the initialization of Flow. That's why all configuration is cached by default, the configuration manager will compile all loaded configuration into a PHP file which will be loaded in subsequent calls instead of parsing the YAML files again.

Changes to the configuration are detected and the cache is flushed when needed. In order to flush caches manually (should that be needed), use the following command:

```
$ ./flow flow:cache:flush
```

Configuration Validation

Errors in configuration can lead to hard to spot errors and seemingly random weird behavior. Flow therefore comes with a general purpose array validator which can check PHP arrays for validity according to some schema.

This validator is used in the `configuration:validate` command:

```
$ ./flow configuration:validate --type Settings
Validating configuration for type: "Settings"

16 schema files were found:
 - package:"Neos.Flow" schema:"Settings/Neos.Flow.aop" -> is valid
...
 - package:"Neos.Flow" schema:"Settings/Neos.Flow.utility" -> is valid

The configuration is valid!
```

See the command help for details on how to use the validation.

Writing Schemata

The schema format is adapted from the JSON Schema standard; currently the Parts 5.1 to 5.25 of the json-schema specification are implemented, with the following deviations from the specification:

- The "type" constraint is required for all properties.
- The validator only executes the checks that make sense for a specific type, see list of possible constraints below.
- The "format" constraint for string type has additional class-name and instance-name options.
- The "dependencies" constraint of the spec is not implemented.
- Similar to "patternProperties" "formatProperties" can be specified specified for dictionaries

> **Warning:** While the *configuration:validate* command will stay like it is, the inner workings of the schema validation are still subject to change. The location of schema files and the syntax might be adjusted in the future, as we (and you) gather real-world experience with this.
>
> With that out of the way: feel free to create custom schemata and let us know of any issues you find or suggestion you have!

The schemas are searched in the path *Resources/Private/Schema* of all active Packages. The schema-filenames must match the pattern `<type>.<path>.schema.yaml`. The type and/or the path can also be expressed as subdirectories of *Resources/Private/Schema*. So *Settings/Neos/Flow.persistence.schema.yaml* will match the same paths as *Settings.Neos.Flow.persistence.schema.yaml* or *Settings/Neos.Flow/persistence.schema.yaml*.

Here is an example of a schema, from *Neos.Flow.core.schema.yaml*:

```
type: dictionary
additionalProperties: FALSE
properties:
  'context': { type: string, required: TRUE }
  'phpBinaryPathAndFilename': { type: string, required: TRUE }
```

It declares the constraints for the *Neos.Flow.core* setting:

- the setting is a dictionary (an associative array in PHP nomenclature)
- properties not defined in the schema are not not allowed
- the properties `context` and `phpBinaryPathAndFilename` are both required and of type string

General constraints for all types (for implementation see `validate` method in `SchemaValidator`):

- type
- disallow
- enum

Additional constraints allowed per type:

string pattern, minLength, maxLength, format(date-time|date|time|uri|email|ipv4|ipv6|ip-address|host-name|class-name|interface-name)

number maximum, minimum, exclusiveMinimum, exclusiveMaximum, divisibleBy

integer maximum, minimum, exclusiveMinimum, exclusiveMaximum, divisibleBy

boolean –

array minItems, maxItems, items

dictionary properties, patternProperties, formatProperties, additionalProperties

null –

any –

2.3.5 Object Framework

The lifecycle of objects are managed centrally by the object framework. It offers convenient support for Dependency Injection and provides some additional features such as a caching mechanism for objects. Because all packages are built on this foundation it is important to understand the general concept of objects in Flow. Note, the object management features of Flow are by default only enabled for classes in packages belonging to one of the *neos-*`* package types. All other classes are not considered by default. If you need that (see *Enabling Other Package Classes For Object Management*).

Tip: A very good start to understand the idea of Inversion of Control and Dependency Injection is reading Martin Fowler's article on the topic.

Creating Objects

In simple, self-contained applications, creating objects is as simple as using the `new` operator. However, as the program gets more complex, a developer is confronted with solving dependencies to other objects, make classes configurable (maybe through a factory method) and finally assure a certain scope for the object (such as `Singleton` or `Prototype`). Howard Lewis Ship explained this circumstances nicely in his blog (quite some time ago):

> Once you start thinking in terms of large numbers of objects, and a whole lot of just in time object creation and configuration, the question of *how* to create a new object doesn't change (that's what `new` is for) ... but the questions *when* and *who* become difficult to tackle. Especially when the *when* is very dynamic, due to just-in-time instantiation, and the *who* is unknown, because there are so many places a particular object may be used.

The Object Manager is responsible for object building and dependency resolution (we'll discover shortly why dependency injection makes such a difference to your application design). In order to fulfill its task, it is important that all objects are instantiated only through the object framework.

Important: As a general rule of thumb for those not developing the Flow core itself but your very own packages:

Use Dependency Injection whenever possible for retrieving singletons.

Object Scopes

Objects live in a specific scope. The most commonly used are *prototype* and *singleton*:

Scope	Description
singleton	The object instance is unique during one request - each injection by the Object Manager or explicit call of `get()` returns the same instance. A request can be an HTTP request or a run initiated from the command line.
prototype (default)	The object instance is not unique - each injection or call of the Object Factory's `create` method returns a fresh instance.
session	The object instance is unique during the whole user session - each injection or `get()` call returns the same instance.

Background: Objects in PHP

In PHP, objects of the scope `prototype` are created with the `new` operator:

```
$myFreshObject = new \MyCompany\MyPackage\MyClassName();
```

In contrast to Prototype, the Singleton design pattern ensures that only one instance of a class exists at a time. In PHP the Singleton pattern is often implemented by providing a static function (usually called `getInstance`), which returns a unique instance of the class:

```
/**
 * Implementation of the Singleton pattern
 */
class ASingletonClass {

        protected static $instance;

        static public function getInstance() {
                if (!is_object(self::$instance)) {
                        self::$instance = $this;
                }
                return self::$instance;
        }
}
```

Although this way of implementing the singleton will possibly not conflict with the Object Manager, it is counterproductive to the integrity of the system and might raise problems with unit testing (sometimes Singleton is referred to as an *Anti Pattern*). The above examples are *not recommended* for the use within Flow applications.

The scope of an object is determined from its configuration (see also *Configuring objects*). The recommended way to specify the scope is the `@scope` annotation:

```
      namespace MyCompany\MyPackage;

use Neos\Flow\Annotations as Flow;

      /**
       * A sample class
       *
       * @Flow\Scope("singleton")
```

(continues on next page)

(continued from previous page)

```
     */
    class SomeClass {
    }
```

Prototype is the default scope and is therefore assumed if no `@scope` annotation or other configuration was found.

Creating Prototypes

To create prototype objects, just use the `new` operator as you are used to:

```
$myFreshObject = new \MyCompany\MyPackage\MyClassName();
```

When you do this, some magic is going on behind the scenes which still makes sure the object you get back is managed by the object framework. Thus, all dependencies are properly injected into the object, lifecycle callbacks are fired, and you can use Aspect-Oriented Programming, etc.

Behind the scenes of the Object Framework

In order to provide the functionality that you can just use `new` to create new prototype objects, a lot of advanced things happen behind the scenes.

Flow internally copies all classes to another file, and appends `_Original` to their class name. Then, it creates a new class under the original name where all the magic is happening.

However, you as a user do not have to deal with that. The only thing you need to remember is using `new` for creating new Prototype objects. And you might know this from PHP ;-)

Retrieving Singletons

The Object Manager maintains a registry of all instantiated singletons and ensures that only one instance of each class exists. The preferred way to retrieve a singleton object is dependency injection.

Example: Retrieving the Object Manager through dependency injection

```
namespace MyCompany\MyPackage;

/**
 * A sample class
 */
class SampleClass {

        /**
         * @var \Neos\Flow\ObjectManagement\ObjectManagerInterface
         */
        protected $objectManager;

        /**
         * Constructor.
         * The Object Manager will automatically be passed (injected) by the object
         * framework on instantiating this class.
         *
         * @param \Neos\Flow\ObjectManagement\ObjectManagerInterface $objectManager
         */
```

(continues on next page)

(continued from previous page)

```
        public function __construct(\Neos\Flow\ObjectManagement\
↪ObjectManagerInterface $objectManager) {
                $this->objectManager = $objectManager;
        }
}
```

Once the `SampleClass` is being instantiated, the object framework will automagically pass a reference to the Object Manager (which is an object of scope *singleton*) as an argument to the constructor. This kind of dependency injection is called *Constructor Injection* and will be explained - together with other kinds of injection - in one of the later sections.

Although dependency injection is what you should strive for, it might happen that you need to retrieve object instances directly. The `ObjectManager` provides methods for retrieving object instances for these rare situations. First, you need an instance of the `ObjectManager` itself, again by taking advantage of constructor injection:

```
public function __construct(\Neos\Flow\ObjectManagement\ObjectManagerInterface
↪$objectManager) {
        $this->objectManager = $objectManager;
}
```

Note: In the text, we commonly refer to the `ObjectManager`. However, in your code, you should always use the `ObjectManagerInterface` if you need an instance of the Object Manager injected.

To explicitly retrieve an object instance use the `get()` method:

```
$myObjectInstance = $objectManager->get('MyCompany\MyPackage\MyClassName');
```

It is *not* possible to pass arguments to the constructor of the object, as the object might be already instantiated when you call `get()`. If the object needs constructor arguments, these must be *configured in Objects.yaml*.

Lifecycle methods

The lifecycle of an object goes through different stages. It boils down to the following order:

1. Solve dependencies for constructor injection
2. Create an instance of the object class, injecting the constructor dependencies
3. Solve and inject dependencies for setter injection
4. Live a happy object-life and solve exciting tasks
5. Dispose the object instance

Your object might want to take some action after certain of the above steps. Whenever one of the following methods exists in the object class, it will be invoked after the related lifecycle step:

1. No action after this step
2. During instantiation the function `__construct()` is called (by PHP itself), dependencies are passed to the constructor arguments
3. After all dependencies have been injected (through constructor- or setter injection) the object's `initializeObject()` method is called. The name of this method is configurable inside *Objects.yaml*. `initializeObject()` is also called if no dependencies were injected.
4. During the life of an object no special lifecycle methods are called

5. Before destruction of the object, the function `shutdownObject()` is called. The name of this method is also configurable.
6. On disposal, the function `__destruct()` is called (by PHP itself)

We strongly recommend that you use the `shutdownObject` method instead of PHP's `__destruct` method for shutting down your object. If you used `__destruct` it might happen that important parts of the framework are already unavailable. Here's a simple example with all kinds of lifecycle methods:

Example: Sample class with lifecycle methods

```
class Foo {

        protected $bar;
        protected $identifier = 'Untitled';

        public function __construct() {
                echo ('Constructing object ...');
        }

        public function injectBar(\MyCompany\MyPackage\BarInterface $bar) {
                $this->bar = $bar;
        }

        public function setIdentifier($identifier) {
                $this->identifier = $identifier;
        }

        public function initializeObject() {
                echo ('Initializing object ...');
        }

        public function shutdownObject() {
                echo ('Shutting down object ...')
        }

        public function __destruct() {
                echo ('Destructing object ...');
        }
}
```

Output:

```
Constructing object ...
Initializing object ...
Shutting down object ...
Destructing object ...
```

Object Registration and API

Object Framework API

The object framework provides a lean API for registering, configuring and retrieving instances of objects. Some of the methods provided are exclusively used within Flow package or in test cases and should possibly not be used elsewhere. By offering Dependency Injection, the object framework helps you to avoid creating rigid interdependencies between objects and allows for writing code which is hardly or even not at all aware of the framework it is working in. Calls to the Object Manager should therefore be the exception.

For a list of available methods please refer to the API documentation of the interface `Neos\Flow\ObjectManagement\ObjectManagerInterface`.

Object Names vs. Class Names

We first need to introduce some namings: A *class name* is the name of a PHP class, while an *object name* is an identifier which is used inside the object framework to identify a certain object.

By default, the *object name* is identical to the PHP class which contains the object's code. A class called `MyCompany\MyPackage\MyImplementation` will be automatically available as an object with the exact same name. Every part of the system which asks for an object with a certain name will therefore - by default - get an instance of the class of that name.

It is possible to replace the original implementation of an object by another one. In that case the class name of the new implementation will naturally differ from the object name which stays the same at all times. In these cases it is important to be aware of the fine difference between an *object name* and a *class name*.

All PHP interfaces for which only one implementation class exist are also automatically registered as *object names*, with the implementation class being returned when asked for an instance of the interface.

Thus, you can also ask for interface implementations:

```
$objectTypeInstance = $objectManager->get('MyCompany\MyPackage\MyInterface');
```

Note: If zero or more than one class implements the interface, the Object Manager will throw an exception.

The advantage of programming against interfaces is the increased flexibility: By referring to interfaces rather than classes it is possible to write code depending on other classes without the need to be specific about the implementation. Which implementation will actually be used can be set at a later point in time by simple means of configuration.

With Flow version 6.2 it's also possible to use "virtual object names" that don't represent an interface or class name (see *Virtual Objects*).

Object Dependencies

The intention to base an application on a combination of packages and objects is to force a clean separation of domains which are realized by dedicated objects. The less each object knows about the internals of another object, the easier it is to modify or replace one of them, which in turn makes the whole system flexible. In a perfect world, each of the objects could be reused in a variety of contexts, for example independently from certain packages and maybe even outside the Flow framework.

Dependency Injection

An important prerequisite for reusable code is already met by encouraging encapsulation through object orientation. However, the objects are still aware of their environment as they need to actively collaborate with other objects and the framework itself: An authentication object will need a logger for logging intrusion attempts and the code of a shop system hopefully consists of more than just one class. Whenever an object refers to another directly, it adds more complexity and removes flexibility by opening new interdependencies. It is very difficult or even impossible to reuse such hardwired classes and testing them becomes a nightmare.

By introducing *Dependency Injection*, these interdependencies are minimized by inverting the control over resolving the dependencies: Instead of asking for the instance of an object actively, the depending object just gets one *injected* by the Object Manager. This methodology is also referred to as the "Hollywood Principle": *Don't call us, we'll call you.* It helps in the development of code with loose coupling and high cohesion — or in short: It makes you a better programmer.

In the context of the previous example it means that the authentication object announces that it needs a logger which implements a certain PHP interface (for example the `Psr\Log\LoggerInterface`). The object itself has no control over what kind of logger (file-logger, sms-logger, ...) it finally gets and it doesn't have to care about it anyway as long as it matches the expected API. As soon as the authentication object is instantiated, the object manager will resolve these dependencies, prepare an instance of a logger and inject it to the authentication object.

Reading Tip

An article by Jonathan Amsterdam discusses the difference between creating an object and requesting one (i.e. using `new` versus using dependency injection). It demonstrates why `new` should be considered as a low-level tool and outlines issues with polymorphism. He doesn't mention dependency injection though ...

Dependencies on other objects can be declared in the object's configuration (see *Configuring objects*) or they can be solved automatically (so called autowiring). Generally there are two modes of dependency injection supported by Flow: *Constructor Injection* and *Setter Injection*.

Note: Please note that Flow removes all injected properties before serializing an object. Then after unserializing injections happen again. That means that injected properties are fresh instances and do not keep any state from before the serialization. That hold true also for Prototypes. If you want to keep a Prototype instance with its state throughout a serialize/unserialize cycle you should not inject the Prototype but rather create it in constructor of the object.

Constructor Injection

With constructor injection, the dependencies are passed as constructor arguments to the depending object while it is instantiated. Here is an example of an object `Foo` which depends on an object `Bar`:

Example: A simple example for Constructor Injection

```
namespace MyCompany\MyPackage;

class Foo {

        protected $bar;

        public function __construct(\MyCompany\MyPackage\BarInterface $bar) {
                $this->bar = $bar;
        }
```

(continues on next page)

(continued from previous page)

```
        public function doSomething() {
                $this->bar->doSomethingElse();
        }
}
```

So far there's nothing special about this class, the type hint just makes sure that an instance of a class implementing the `\MyCompany\MyPackage\BarInterface` is passed to the constructor. However, this is already a quite flexible approach because the type of `$bar` can be determined from outside by just passing one or the another implementation to the constructor.

Now the Flow Object Manager does some magic: By a mechanism called *Autowiring* all dependencies which were declared in a constructor will be injected automagically if the constructor argument provides a type definition (i.e. `\MyCompany\MyPackage\BarInterface` in the above example). Autowiring is activated by default (but can be switched off), therefore all you have to do is to write your constructor method.

The object framework can also be configured manually to inject a certain object or object type. You'll have to do that either if you want to switch off autowiring or want to specify a configuration which differs from would be done automatically.

Example: Objects.yaml file for Constructor Injection

```
MyCompany\MyPackage\Foo:
  arguments:
    1:
      object: 'MyCompany\MyPackage\Bar'
```

The three lines above define that an object instance of `\MyCompany\MyPackage\Bar` must be passed to the first argument of the constructor when an instance of the object `MyCompany\MyPackage\Foo` is created.

Setter Injection

With setter injection, the dependencies are passed by calling *setter methods* of the depending object right after it has been instantiated. Here is an example of the `Foo` class which depends on a `Bar` object - this time with setter injection:

Example: A simple example for Setter Injection

```
namespace MyCompany\MyPackage;

class Foo {

        protected $bar;

        public function setBar(\MyCompany\MyPackage\BarInterface $bar) {
                $this->bar = $bar;
        }

        public function doSomething() {
                $this->bar->doSomethingElse();
        }
}
```

Analog to the constructor injection example, a `BarInterface` compatible object is injected into the `Foo` object. In this case, however, the injection only takes place after the class has been instantiated and a possible constructor method has been called. The necessary configuration for the above example looks like this:

Example: Objects.yaml file for Setter Injection

```
MyCompany\MyPackage\Foo:
  properties:
    bar:
      object: 'MyCompany\MyPackage\BarInterface'
```

Unlike constructor injection, setter injection like in the above example does not offer the autowiring feature. All dependencies have to be declared explicitly in the object configuration.

To save you from writing large configuration files, Flow supports a second type of setter methods: By convention all methods whose name start with `inject` are considered as setters for setter injection. For those methods no further configuration is necessary, dependencies will be autowired (if autowiring is not disabled):

Example: The preferred way of Setter Injection, using an inject method

```
namespace MyCompany\MyPackage;

class Foo {

        protected $bar;

        public function injectBar(\MyCompany\MyPackage\BarInterface $bar) {
                $this->bar = $bar;
        }

        public function doSomething() {
                $this->bar->doSomethingElse();
        }
}
```

Note the new method name `injectBar` - for the above example no further configuration is required. Using `inject*` methods is the preferred way for setter injection in Flow.

Note: If both, a `set*` and an `inject*` method exist for the same property, the `inject*` method has precedence.

Constructor- or Setter Injection?

The natural question which arises at this point is *Should I use constructor- or setter injection?* There is no answer across-the-board — it mainly depends on the situation and your preferences. The authors of the Java-based Spring Framework for example prefer Setter Injection for its flexibility. The more puristic developers of PicoContainer strongly plead for using Constructor Injection for its cleaner approach. Reasons speaking in favor of constructor injections are:

- Constructor Injection makes a stronger dependency contract
- It enforces a determinate state of the depending object: using setter Injection, the injected object is only available after the constructor has been called

However, there might be situations in which constructor injection is not possible or even cumbersome:

- If an object has many dependencies and maybe even many optional dependencies, setter injection is a better solution.
- Subclasses are not always in control over the arguments passed to the constructor or might even be incapable of overriding the original constructor. Then setter injection is your only chance to get dependencies injected.
- Setter injection can be helpful to avoid circular dependencies between objects.

- Setters provide more flexibility to unit tests than a fixed set of constructor arguments

Property Injection

Setter injection is the academic, clean way to set dependencies from outside. However, writing these setters can become quite tiresome if all they do is setting the property. For these cases Flow provides support for *Property Injection*:

Example: Example for Property Injection

```
	namespace MyCompany\MyPackage;

use Neos\Flow\Annotations as Flow;

	class Foo {

		/**
		 * An instance of a BarInterface compatible object.
		 *
		 * @var \MyCompany\MyPackage\BarInterface
		 * @Flow\Inject
		 */
		protected $bar;

		public function doSomething() {
			$this->bar->doSomethingElse();
		}
	}
```

You could say that property injection is the same like setter injection — just without the setter. The `Inject` annotation tells the object framework that the property is supposed to be injected and the `@var` annotation specifies the type. Note that property injection even works (and should only be used) with protected properties. The *Objects.yaml* configuration for property injection is identical to the setter injection configuration.

Note: If a setter method exists for the same property, it has precedence.

Setting properties directly, without a setter method, surely is convenient - but is it clean enough? In general it is a bad idea to allow direct access to mutable properties because you never know if at some point you need to take some action while a property is set. And if thousands of users (or only five) use your API, it's hard to change your design decision in favor of a setter method.

However, we don't consider injection methods as part of the public API. As you've seen, Flow takes care of all the object dependencies and the only other code working with injection methods directly are unit tests. Therefore we consider it safe to say that you can still switch back from property injection to setter injection without problems if it turns out that you really need it.

Lazy Dependency Injection

Using Property Injection is, in its current implementation, the most performant way to inject a dependency. As an important additional benefit you also get Lazy Dependency Injection: instead of loading the class of the dependency, instantiating and intializing it, a `proxy` is injected instead. This object waits until it will be accessed the first time. Once you start using the dependency, the proxy will build or retrieve the real dependency, call the requested method and return the result. On all following method calls, the real object will be used.

By default all dependencies injected through Property Injection are lazy. Usually this process is fully transparent to the user, unless you start passing around dependencies to other objects:

Example: Passing a dependency around

```
	namespace MyCompany\MyPackage;

use Neos\Flow\Annotations as Flow;

	class Foo {

		/**
		 * A dependency, injected lazily:
		 *
		 * @var \MyCompany\MyPackage\BarInterface
		 * @Flow\Inject
		 */
		protected $bar;

		...

		public function doSomething() {
			$this->baz->doSomethingElse($this->bar);
		}

	}

	class Baz {

		public function doSomethingElse(Bar $bar) {
			...
		}

	}
```

The above example will break: at the time you pass `$this->bar` to the `doSomethingElse()` method, it is not yet a `Bar` object but a `DependencyProxy` object. Because `doSomethingElse()` has a type hint requiring a `Bar` object, PHP will issue a fatal error.

There are two ways to solve this:

- activating the dependency manually
- turning off lazy dependency injection for this property

Example: Manually activating a dependency

```
	namespace MyCompany\MyPackage;

use Neos\Flow\Annotations as Flow;

```

(continues on next page)

(continued from previous page)

```
	class Foo {

		/**
		 * A dependency, injected lazily:
		 *
		 * @var \MyCompany\MyPackage\BarInterface
		 * @Flow\Inject
		 */
		protected $bar;

		...

		public function doSomething() {
			if ($this->bar instanceof \Neos\Flow\ObjectManagement\
↪DependencyInjection\DependencyProxy) {
				$this->bar->_activateDependency();
			}
			$this->baz->doSomethingElse($this->bar);
		}

	}
```

In the example above, `$this->bar` is activated before it is passed to the next method. It's important to check if the object still is a proxy because otherwise calling `_activateDependency()` will fail.

Example: Turning off lazy dependency injection

```
	namespace MyCompany\MyPackage;

use Neos\Flow\Annotations as Flow;

	class Foo {

		/**
		 * A dependency, injected eagerly
		 *
		 * @var \MyCompany\MyPackage\BarInterface
		 * @Flow\Inject(lazy = FALSE)
		 */
		protected $bar;

		...

		public function doSomething() {
			$this->baz->doSomethingElse($this->bar);
		}

	}
```

In the second solution, lazy dependency injection is turned off. This way you can be sure that `$this->bar` always contains the object you expected, but you don't benefit from the speed optimizations.

Settings Injection

No, this headline is not misspelled. Flow offers some convenient feature which allows for automagically injecting the settings of the current package without the need to configure the injection. If a class contains a method called `injectSettings` and autowiring is not disabled for that object, the Object Builder will retrieve the settings of the package the object belongs to and pass it to the `injectSettings` method.

Example: the magic injectSettings method

```
namespace MyCompany\MyPackage;

class Foo {

        protected $settings = array();

        public function injectSettings(array $settings) {
                $this->settings = $settings;
        }

        public function doSomething() {
                var_dump($this->settings);
        }
}
```

The `doSomething` method will output the settings of the `MyPackage` package.

In case you only need a specific setting, there's an even more convenient way to inject a single setting value into a class property:

```
      namespace Acme\Demo;

use Neos\Flow\Annotations as Flow;

      class SomeClass {

              /**
               * @var string
               * @Flow\InjectConfiguration("administrator.name")
               */
              protected $name;

              /**
               * @var string
               * @Flow\InjectConfiguration(path="email", package="SomeOther.Package")
               */
              protected $emailAddress;

      }
```

The `InjectConfiguration` annotation also supports for injecting all settings of a package. And it can also be used to inject any other registered configuration type:

```
namespace Acme\Demo;

class SomeClass {

        /**
         * @var array
```

(continues on next page)

(continued from previous page)

```
         * @Flow\InjectConfiguration(package="SomeOther.Package")
         */
        protected $allSettingsOfSomeOtherPackage;

        /**
         * @var array
         * @Flow\InjectConfiguration(type="Views")
         */
        protected $viewsConfiguration;

}
```

Required Dependencies

All dependencies defined in a constructor are, by its nature, required. If a dependency can't be solved by autowiring or by configuration, Flow's object builder will throw an exception.

Also *autowired setter-injected dependencies* are, by default, required. If the object builder can't autowire an object for an injection method, it will throw an exception.

Dependency Resolution

The dependencies between objects are only resolved during the instantiation process. Whenever a new instance of an object class needs to be created, the object configuration is checked for possible dependencies. If there is any, the required objects are built and only if all dependencies could be resolved, the object class is finally instantiated and the dependency injection takes place.

During the resolution of dependencies it might happen that circular dependencies occur. If an object `A` requires an object `B` to be injected to its constructor and then again object `B` requires an object `A` likewise passed as a constructor argument, none of the two classes can be instantiated due to the mutual dependency. Although it is technically possible (albeit quite complex) to solve this type of reference, Flow's policy is not to allow circular constructor dependencies at all. As a workaround you can use setter injection instead for either one or both of the objects causing the trouble.

Configuring objects

The behavior of objects significantly depends on their configuration. During the initialization process all classes found in the various *Classes/* directories are registered as objects and an initial configuration is prepared. In a second step, other configuration sources are queried for additional configuration options. Definitions found at these sources are added to the base configuration in the following order:

- If they exist, the *<PackageName>/Configuration/Objects.yaml* will be included.
- Additional configuration defined in the global *Configuration/Objects.yaml* directory is applied.
- Additional configuration defined in the global *Configuration/<ApplicationScope>/Objects.yaml* directory is applied.

Currently there are three important situations in which you want to configure objects:

- Override one object implementation with another
- Set the active implementation for an object type
- Explicitly define and configure dependencies to other objects

Configuring Objects Through Objects.yaml

If a file named *Objects.yaml* exists in the *Configuration* directory of a package, it will be included during the configuration process. The YAML file should stick to Flow's general rules for YAML-based configuration.

Example: Sample Objects.yaml file

```
#                                                                             #
# Object Configuration for the MyPackage package                              #
#                                                                             #

# @package MyPackage

MyCompany\MyPackage\Foo:
  arguments:
    1:
      object: 'MyCompany\MyPackage\Baz'
    2:
      value: "some string"
    3:
      value: false
  properties:
    bar:
      object: 'MyCompany\MyPackage\BarInterface'
    enableCache:
      setting: MyPackage.Cache.enable
```

Configuring Objects Through Annotations

A very convenient way to configure certain aspects of objects are annotations. You write down the configuration directly where it takes effect: in the class file. However, this way of configuring objects is not really flexible, as it is hard coded. That's why only those options can be set through annotations which are part of the class design and won't change afterwards. Currently `scope`, `inject` and `autowiring` are the only supported annotations.

It's up to you defining the scope in the class directly or doing it in a *Objects.yaml* configuration file – both have the same effect. We recommend using annotations in this case, as the scope usually is a design decision which is very unlikely to be changed.

Example: Sample scope annotation

```
/**
 * This is my great class.
 *
 * @Flow\Scope("singleton")
 */
class SomeClass {

}
```

Example: Sample autowiring annotation for a class

```
/**
 * This turns off autowiring for the whole class:
 *
 * @Flow\Autowiring(false)
 */
```

(continues on next page)

(continued from previous page)

```
class SomeClass {

}
```

Example: Sample autowiring annotation for a method

```
/**
 * This turns off autowiring for a single method:
 *
 * @param \Neos\Foo\Bar $bar
 * @Flow\Autowiring(false)
 */
public function injectMySpecialDependency(\Neos\Foo\Bar $bar) {

}
```

Overriding Object Implementations

One advantage of componentry is the ability to replace objects by others without any bad impact on those parts depending on them.

A prerequisite for replaceable objects is that their classes implement a common interface which defines the public API of the original object. Other objects which implement the same interface can then act as a true replacement for the original object without the need to change code anywhere in the system. If this requirement is met, the only necessary step to replace the original implementation with a substitute is to alter the object configuration and set the class name to the new implementation.

To illustrate this circumstance, consider the following classes.

Example: The Greeter object type

```
namespace MyCompany\MyPackage;

interface GreeterInterface {
        public function sayHelloTo($name);
}

class Greeter implements GreeterInterface {
        public function sayHelloTo($name) {
                echo 'Hello ' . $name;
        }
}
```

During initialization the above `Greeter` class will automatically be registered as the default implementation of `MyCompany\MyPackage\GreeterInterface` and is available to other objects. In the class code of another object you might find the following lines.

Example: Using the Greeter object type

```
// Use setter injection for fetching an instance
// of \MyCompany\MyPackage\GreeterInterface:
public function injectGreeter(\MyCompany\MyPackage\GreeterInterface $greeter) {
        $this->greeter = $greeter;
}
```

(continues on next page)

(continued from previous page)

```
public function someAction() {
        $this->greeter->sayHelloTo('Heike');
}
```

If we want to use the much better object `\Neos\OtherPackage\GreeterWithCompliments`, the solution is to let the new implementation implement the same interface.

Example: The improved Greeter object type

```
namespace Neos\OtherPackage;

class GreeterWithCompliments implements \MyCompany\MyPackage\GreeterInterface {
        public function sayHelloTo($name) {
                echo('Hello ' . $name . '! You look so great!');
        }
}
```

Then we have to set which implementation of the `MyCompany\MyPackage\GreeterInterface` should be active and are done:

Example: Objects.yaml file for object type definition

```
MyCompany\MyPackage\GreeterInterface:
  className: 'Neos\OtherPackage\GreeterWithCompliments'
```

The the same code as above will get the improved `GreeterWithCompliments` instead of the simple `Greeter` now.

Configuring Injection

The object framework allows for injection of straight values, objects (i.e. dependencies) or settings either by passing them as constructor arguments during instantiation of the object class or by calling a setter method which sets the wished property accordingly. The necessary configuration for injecting objects is usually generated automatically by the *autowiring* capabilities of the Object Builder. Injection of straight values or settings, however, requires some explicit configuration.

Injection Values

Regardless of what injection type is used (constructor or setter injection), there are three kinds of value which can be injected:

- *value*: static value of a simple type. Can be string, integer, boolean or array and is passed on as is.
- *object*: object name which represents a dependency. Dependencies of the injected object are resolved and an instance of the object is passed along.
- *setting*: setting defined in one of the *Settings.yaml* files. A path separated by dots specifies which setting to inject.

Constructor Injection

Arguments for constructor injection are defined through the *arguments* option. Each argument is identified by its position, counting starts with 1.

Example: Sample class for Constructor Injection

```
namespace MyCompany\MyPackage;

class Foo {

        protected $bar;
        protected $identifier;
        protected $enableCache;

        public function __construct(\MyCompany\MyPackage\BarInterface $bar,
↪$identifier,
                    $enableCache) {
                $this->bar = $bar;
                $this->identifier = $identifier;
                $this->enableCache = $enableCache;
        }

        public function doSomething() {
                $this->bar->doSomethingElse();
        }
}
```

Example: Sample configuration for Constructor Injection

```
MyCompany\MyPackage\Foo:
  arguments:
    1:
      object: 'MyCompany\MyPackage\Bar'
    2:
      value: "some string"
    3:
      setting: "MyPackage.Cache.enable"
```

Note: It is usually not necessary to configure injection of objects explicitly. It is much more convenient to just declare the type of the constructor arguments (like `MyCompany\MyPackage\BarInterface` in the above example) and let the autowiring feature configure and resolve the dependencies for you.

Setter Injection

The following class and the related *Objects.yaml* file demonstrate the syntax for the definition of setter injection:

Example: Sample class for Setter Injection

```
namespace MyCompany\MyPackage;

class Foo {

        protected $bar;
```

(continues on next page)

(continued from previous page)

```
        protected $identifier = 'Untitled';
        protected $enableCache = FALSE;

        public function injectBar(\MyCompany\MyPackage\BarInterface $bar) {
                $this->bar = $bar;
        }

        public function setIdentifier($identifier) {
                $this->identifier = $identifier;
        }

        public function setEnableCache($enableCache) {
                $this->enableCache = $enableCache;
        }

        public function doSomething() {
                $this->bar->doSomethingElse();
        }
}
```

Example: Sample configuration for Setter Injection

```
MyCompany\MyPackage\Foo:
  properties:
    bar:
      object: 'MyCompany\MyPackage\Bar'
    identifier:
      value: 'some string'
    enableCache:
      setting: 'MyPackage.Cache.enable'
```

As you can see, it is important that a setter method with the same name as the property, preceded by `inject` or `set` exists. It doesn't matter though, if you choose `inject` or `set`, except that `inject` has the advantage of being autowireable. As a rule of thumb we recommend using `inject` for required dependencies and values and `set` for optional properties.

Injection of Objects Specified in Settings

In some cases it might be convenient to specify the name of the object to be injected in the *settings* rather than in the objects configuration. This can be achieved by specifying the settings path instead of the object name:

Example: Injecting an object specified in the settings

```
MyCompany\MyPackage\Foo:
  properties:
    bar:
      object: 'MyCompany.MyPackage.fooStuff.barImplementation'
```

Example: Settings.yaml of MyPackage

```
MyCompany:
  MyPackage:
    fooStuff:
      barImplementation: 'MyCompany\MyPackage\Bars\ASpecialBar'
```

Nested Object Configuration

While autowiring and automatic dependency injection offers a great deal of convenience, it is sometimes necessary to have a fine grained control over which objects are injected with which third objects injected.

Consider a Flow cache object, a `VariableCache` for example: the cache itself depends on a cache backend which on its part requires a few settings passed to its constructor - this readily prepared cache should now be injected into another object. Sounds complex? With the objects configuration it is however possible to configure even that nested object structure:

Example: Nesting object configuration

```
MyCompany\MyPackage\Controller\StandardController:
  properties:
    cache:
      object:
        name: 'Neos\Cache\VariableCache'
        arguments:
          1:
            value: MyCache
          2:
            object:
              name: 'Neos\Cache\Backend\File'
              properties:
                cacheDirectory:
                  value: /tmp/
```

Disabling Autowiring

Injecting dependencies is a common task. Because Flow can detect the type of dependencies a constructor needs, it automatically configures the object to ensure that the necessary objects are injected. This automation is called *autowiring* and is enabled by default for every object. As long as autowiring is in effect, the Object Builder will try to autowire all constructor arguments and all methods named after the pattern `inject*`.

If, for some reason, autowiring is not wanted, it can be disabled by setting an option in the object configuration:

Example: Turning off autowiring support in Objects.yaml

```
MyCompany\MyPackage\MyObject:
  autowiring: false
```

Autowiring can also be switched off through the `@Flow\Autowiring(false)` annotation - either in the documentation block of a whole class or of a single method. For the latter the annotation only has an effect when used in comment blocks of a constructor or of a method whose name starts with `inject`.

Custom Factories

Complex objects might require a custom factory which takes care of all important settings and dependencies. As we have seen previously, a logger consists of a frontend, a backend and configuration options for that backend. Instead of creating and configuring these objects on your own, you should use the `Neos\Flow\Log\PsrLoggerFactory` which provides a convenient `get` method taking care of all the rest:

```
$myCache = $loggerFactory->get('systemLogger');
```

It is possible to specify for each object if it should be created by a custom factory rather than the Object Builder. Consider the following configuration:

Example: Sample configuration for a Custom Factory

```
Neos\Flow\Log\PsrSystemLoggerInterface:
  scope: singleton
  factoryObjectName: Neos\Flow\Log\PsrLoggerFactory
  factoryMethodName: get
```

From now on the LoggerFactory's `get` method will be called each time an object of type `PsrSystemLoggerInterface` needs to be instantiated. If arguments were passed to the `ObjectManagerInterface::get()` method or defined in the configuration, they will be passed through to the custom factory method:

Example: YAML configuration for a Custom Factory with default arguments

```
Neos\Flow\Log\PsrSystemLoggerInterface:
  scope: singleton
  factoryObjectName: Neos\Flow\Log\PsrLoggerFactory
  factoryMethodName: get
  arguments:
    1:
      value: 'systemLogger'
```

Example: YAML configuration for a static custom factory method

```
Acme\Foo\Object:
  scope: prototype
  factoryMethodName: Acme\Foo\ObjectFactory::fromValue
  arguments:
    1:
      settings: 'Acme.Foo.Object.ConfigurableValue'
```

Note that if you only specify the *factoryMethodName*, it needs to be the fully qualified name.

Example: PHP code using the custom factory

```
$myCache = $objectManager->get(\Neos\Flow\Log\PsrSystemLoggerInterface::class);
```

`$objectManager` is a reference to the `Neos\Flow\ObjectManagement\ObjectManager`. The required arguments are automatically built from the values defined in the object configuration.

Name of Lifecycle Methods

The default name of a lifecycle methods is `initializeObject` and `shutdownObject`. If these methods exist, the initialization method will be called after the object has been instantiated or recreated and all dependencies are injected and the shutdown method is called before the Object Manager quits its service.

As the initialization method is being called after creating an object *and* after recreating/reconstituting an object, there are cases where different code should be executed. That is why the initialization method gets a parameter, which is one of the `\Neos\Flow\ObjectManagement\ObjectManagerInterface::INITIALIZATIONCAUSE_*` constants:

`\Neos\Flow\ObjectManagement\ObjectManagerInterface::INITIALIZATIONCAUSE_CREATED`
If the object is newly created (i.e. the constructor has been called)

\Neos\Flow\ObjectManagement\ObjectManagerInterface::INITIALIZATIONCAUSE_RECREATED
If the object has been recreated/reconstituted (i.e. the constructor has not been called)

The name of both methods is configurable per object for situations you don't have control over the name of your initialization method (maybe, because you are integrating legacy code):

Example: Objects.yaml configuration of the initialization and shutdown method

```
MyCompany\MyPackage\MyObject:
  lifecycleInitializationMethod: myInitializeMethodName
  lifecycleShutdownMethod: myShutdownMethodName
```

Static Method Result Compilation

Some part of a Flow application may rely on data which is static during runtime, but which cannot or should not be hardcoded.

One example is the validation rules generated by the MVC framework for arguments of a controller action: the base information (PHP methods for the actions, type hints and arguments of these methods) is static. However, the validation rules should be determined automatically by the framework instead of being configured or hardcoded elsewhere. On the other hand, generating validation rules during runtime unnecessarily slows down the application. The solution is static method result compilation.

A method which generates data based on information already known at compile time can usually be made static. Consider the following example:

```
/**
 * Returns a map of action method names and their parameters.
 *
 * @return array Array of method parameters by action name
 */
public function getActionMethodParameters() {
        $methodParameters = $this->reflectionService->getMethodParameters(get_class(
↪$this), $this->actionMethodName);
        foreach ($methodParameters as $parameterName => $parameterInfo) {
          ...
        }
        return $methodParameters;
}
```

In the example above, `getActionMethodParameters()` returns data needed during runtime which could easily be pre-compiled.

By annotating the method with `@Flow\CompileStatic` and transforming it into a static method which does not depend on runtime services like persistence, security and so on, the performance in production context can be improved:

```
/**
 * Returns a map of action method names and their parameters.
 *
 * @param \Neos\Flow\ObjectManagement\ObjectManagerInterface $objectManager
 * @return array Array of method parameters by action name
 * @Flow\CompileStatic
 */
static protected function getActionMethodParameters($objectManager) {
        $reflectionService = $objectManager->get(\Neos\Flow\Reflection\
↪ReflectionService::class);
```

(continues on next page)

(continued from previous page)

```
        $className = get_called_class();
        $methodParameters = $reflectionService->getMethodParameters($className, get_
↪class_methods($className));
        foreach ($methodParameters as $parameterName => $parameterInfo) {
            ...
        }
        return $methodParameters;
}
```

The results of methods annotated with `CompileStatic` will only be compile in `Production` context. When Flow is started in a different context, the method will be executed during each run.

Enabling Other Package Classes For Object Management

As stated in the beginning of this part, all classes in packages not in one of the `neos-*` types is not recognized for object management by default. If you still want that you can include those classes via configuration in settings. The configuration consists of a map of package keys to arrays of expressions which match classes to be included. In the following example we include all classes of the `Acme.Objects` package:

```
Neos:
  Flow:
    object:
      includeClasses:
        'Acme.Objects' : ['.*']
```

Note: If you use the `includeClasses` setting on a flow package (which is already enabled for object management) then only the classes that match at least one of the filter expressions are going to be object managed. This can also be used to remove classes inside flow packages from object management by specifying a non-matching expression or an empty array.

Note: The static method must except exactly one argument which is the Flow Object Manager. You cannot use a type hint at this point (for the `$objectManager` argument) because the argument passed could actually be a `DependencyProxy` and not the real ObjectManager. Please refer to the section about Lazy Dependency Injection for more information about `DependencyProxy`.

Virtual Objects

With the `Objects.yaml` configuration, the default behavior of classes can be changed globally. Sometimes it can be useful to configure the same class for multiple different use cases. For example two logger implementations that use the same instance but are configured separately.

With Flow version 6.2 and above it's possible to configure "Virtual Objects". The syntax is the same as described above (see *Configuring objects*) with two differences:

- The object name has to contain a colon (to tell it apart from regular object names)
- The `className` configuration is required (since it can't be inferred from the object name)

Example: Objects.yaml for two virtual logger objects

```
'Some.Package:SystemLogger':
  className: Psr\Log\LoggerInterface
  scope: singleton
  factoryObjectName: Neos\Flow\Log\PsrLoggerFactoryInterface
  factoryMethodName: get
  arguments:
    1:
      value: systemLogger

'Some.Package:SecurityLogger':
  className: Psr\Log\LoggerInterface
  scope: singleton
  factoryObjectName: Neos\Flow\Log\PsrLoggerFactoryInterface
  factoryMethodName: get
  arguments:
    1:
      value: securityLogger
```

With those objects configured, the respective loggers can be instantiated via:

```
$systemLogger = $objectManager->get('Some.Package:SystemLogger');
$securityLogger = $objectManager->get('Some.Package:SecurityLogger');
```

Injecting Virtual Objects

To inject a Virtual Object you can simply use the `name` property of the `Inject` annotation to refer to the Virtual Object name:

Example: SomeClass.php

```
class SomeClass {
    /**
     * @Flow\Inject(name="Some.Package:SystemLogger")
     * @var LoggerInterface
     */
    protected $systemLogger;
```

Alternatively you can use constructor- or setter injection with a corresponding configuration:

Example: Objects.yaml

```
# ...
'Some\Package\SomeClass':
  properties:
    'systemLogger':
      object: 'Some.Package:SystemLogger'
```

Example: SomeClass.php

```
class SomeClass {

    public function injectSystemLogger(LoggerInterface $systemLogger): void
    {
        $this->systemLogger = $systemLogger;
    }
```

2.3.6 Persistence

This chapter explains how to use object persistence in Flow. To do this, it focuses on the persistence based on the *Doctrine* 2 ORM first. There is another mechanism available, called *Generic* persistence, which can be used to add your own persistence backends to Flow. It is explained separately later in the chapter.

Note: The *Generic* persistence is deprecated as of Flow 6.0 and will be dropped in Flow 7.0.

Tip: If you have experience with Doctrine 2 already, your knowledge can be applied fully in Flow. If you have not worked with Doctrine 2 in the past, it might be helpful to learn more about it, as that might clear up questions this documentation might leave open.

Introductory Example

Let's look at the following example as an introduction to how Flow handles persistence. We have a domain model of a Blog, consisting of Blog, Post, Comment and Tag objects:

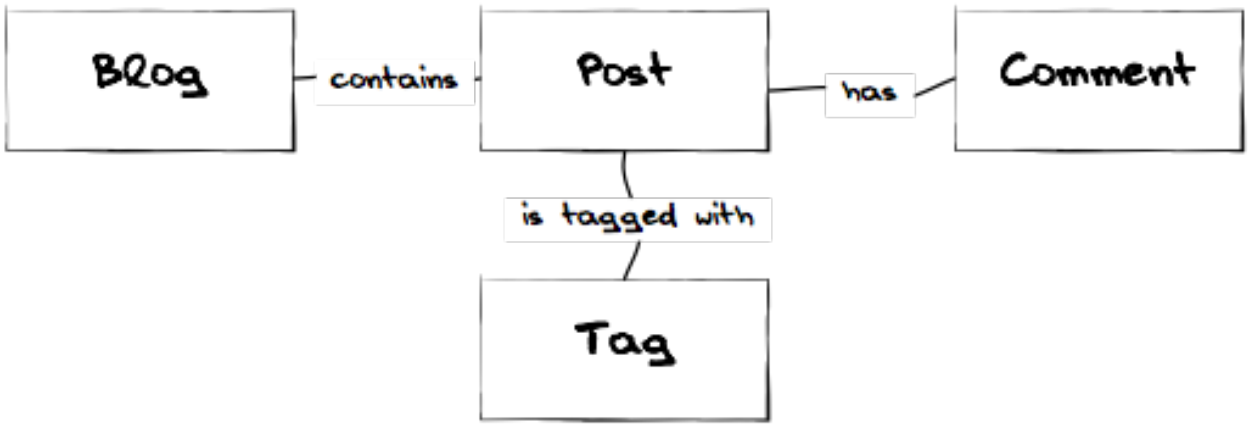

Fig. 24: The objects of the Blog domain model

Connections between those objects are built (mostly) by simple references in PHP, as a look at the `addPost()` method of the `Blog` class shows:

Example: The Blog's addPost() method

```
/**
 * @param \Neos\Blog\Domain\Model\Post $post
 * @return void
 */
public function addPost(\Neos\Blog\Domain\Model\Post $post) {
    $post->setBlog($this);
    $this->posts->add($post);
}
```

The same principles are applied to the rest of the classes, resulting in an object tree of a blog object holding several posts, those in turn having references to their associated comments and tags.

But now we need to make sure the `Blog` and the data in it are still available the next time we need them. In the good old days of programming you might have added some ugly database calls all over the system at this point. In the

currently widespread practice of loving Active Record you'd still add `save()` methods to all or most of your objects. But can it be even easier?

To access an object you need to hold some reference to it. You can get that reference by creating an object or by following some reference to it from some object you already have. This leaves you at a point where you need to find that "first object". This is done by using a *Repository*. A Repository is the librarian of your system, knowing about all the objects it manages. In our model the `Blog` is the entry point to our object tree, so we will add a `BlogRepository`, allowing us to find `Blog` instances by the criteria we need.

Now, before we can find a `Blog`, we need to create and save one. What we do is create the object and `add` it to the `BlogRepository`. This will automagically persist your Blog and you can retrieve it again later.

For all that magic to work as expected, you need to give some hints. This doesn't mean you need to write tons of XML, a few annotations in your code are enough:

Example: Persistence-related annotations in the Blog class

```
namespace Neos\Blog\Domain\Model;

/**
 * A Blog object
 *
 * @Flow\Entity
 */
class Blog {

    /**
     * @var string
     * @Flow\Validate(type="Text")
     * @Flow\Validate(type="StringLength", options={ "minimum"=1, "maximum"=80 })
     * @ORM\Column(length=80)
     */
    protected $title;

    /**
     * @var \Doctrine\Common\Collections\ArrayCollection<\Neos\Blog\Domain\Model\Post>
     * @ORM\OneToMany(mappedBy="blog")
     * @ORM\OrderBy({"date" = "DESC"})
     */
    protected $posts;

    ...

}
```

The first annotation to note is the `Entity` annotation, which tells the persistence framework it needs to persist `Blog` instances if they have been added to a Repository. In the `Blog` class we have some member variables, they are persisted as well by default. The persistence framework knows their types by looking at the `@var` annotation you use anyway when documenting your code (you do document your code, right?).

The *Column* annotation on `$title` is an optimization since we allow only 80 chars anyway. In case of the `$posts` property the persistence framework persists the objects held in that `ArrayCollection` as independent objects in a one-to-many relationship. Apart from those annotations your domain object's code is completely unaware of the persistence infrastructure.

Let's conclude by taking a look at the BlogRepository code:

Example: Code of a simple BlogRepository

```
use Neos\Flow\Annotations as Flow;

      /**
       * A BlogRepository
       *
       * @Flow\Scope("singleton")
       */
      class BlogRepository extends \Neos\Flow\Persistence\Repository {
      }
```

As you can see we get away with very little code by simply extending the Flow-provided repository class, and still we already have methods like `findAll()` and even magic calls like `findOneBy<PropertyName>()` available. If we need some specialized find methods in our repository, we can make use of the query building API:

Example: Using the query building API in a Repository

```
/**
 * A PostRepository
 */
class PostRepository extends \Neos\Flow\Persistence\Repository {

    /**
     * Finds posts by the specified tag and blog
     *
     * @param \Neos\Blog\Domain\Model\Tag $tag
     * @param \Neos\Blog\Domain\Model\Blog $blog The blog the post must refer to
     * @return \Neos\Flow\Persistence\QueryResultInterface The posts
     */
    public function findByTagAndBlog(\Neos\Blog\Domain\Model\Tag $tag,
      \Neos\Blog\Domain\Model\Blog $blog) {
        $query = $this->createQuery();
        return $query->matching(
            $query->logicalAnd(
                $query->equals('blog', $blog),
                $query->contains('tags', $tag)
            )
        )
        ->setOrderings(array(
            'date' => \Neos\Flow\Persistence\QueryInterface::ORDER_DESCENDING)
        )
        ->execute();
    }
}
```

If you like to do things the hard way you can get away with implementing `\Neos\Flow\Persistence\RepositoryInterface` yourself, though that is something the normal developer never has to do.

Note: With the query building API it is possible to query for properties of sub-entities easily via a dot-notation path. When querying multiple properties of a collection property, it is ambiguous if you want to select a single sub-entity with the given matching constraints, or multiple sub-entities which each matching a part of the given constraints.

Since 4.0 Flow will translate such a query to "find all entities where a single sub-entity matches all the constraints", which is the more common case. If you intend a different querying logic, you should fall back to DQL or native SQL queries instead.

Basics of Persistence in Flow

On the Principles of DDD

From Evans, the rules we need to enforce include:

- The root Entity has global identity and is ultimately responsible for checking invariants.
- Root Entities have global identity. Entities inside the boundary have local identity, unique only within the Aggregate.
- Value Objects do not have identity. They are only identified by the combination of their properties and are therefore immutable.
- Nothing outside the Aggregate boundary can hold a reference to anything inside, except to the root Entity. The root Entity can hand references to the internal Entities to other objects, but they can only use them transiently (within a single method or block).
- Only Aggregate Roots can be obtained directly with database queries. Everything else must be done through traversal.
- Objects within the Aggregate can hold references to other Aggregate roots.
- A delete operation must remove everything within the Aggregate boundary all at once.
- When a change to any object within the Aggregate boundary is committed, all invariants of the whole Aggregate must be satisfied.

On the relationship between adding and retrieving

When you `add()` something to a repository and do a `findAll()` immediately afterwards, you might be surprised: the freshly added object will not be found. This is not a bug, but a decision we took on purpose. Here is why.

When you add an object to a repository, it is added to the internal identity map and will be persisted later (when `persistAll()` is called). It is therefore still in a transient state - but all query operations go directly to the underlying data storage, because we need to check that anyway. So instead of trying to query the in-memory objects we decided to ignore transient objects for queries[4].

If you need to query for objects you just created, feel free to have the `PersistenceManager` injected and use `persistAll()` in your code.

How changes are persisted

When you `add` or `remove` an object to or from a repository, the object will be added to or removed from the underlying persistence as expected upon `persistAll`. But what about changes to already persisted objects? As we have seen, those changes are only persisted, if the changed object is given to `update` on the corresponding repository.

Now, for objects that have no corresponding repository, how are changes persisted? In the same way you fetch those objects from their parent - by traversal. Flow follows references from objects managed in a repository (aggregate roots) for all persistence operations, unless the referenced object itself is an aggregate root.

When using the Doctrine 2 persistence, this is done by virtually creating cascade attributes on the mapped associations. That means if you changed an object attached to some aggregate root, you need to hand that aggregate root to `update` for the change to be persisted.

[4] An alternative would have been to do an implicit persist call before a query, but that seemed to be confusing.

Safe request methods are read-only

According to the HTTP 1.1 specification, so called "safe request methods" (usually GET or HEAD requests) should not change your data on the server side and should be considered read-only. If you need to add, modify or remove data, you should use the respective request methods (POST, PUT, DELETE and PATCH).

Flow supports this principle because it helps making your application more secure and perform better. In practice that means for any Flow application: if the current request is a "safe request method", the persistence framework will NOT trigger `persistAll()` at the end of the script run.

You are free to call `PersistenceManager->persistAll()` manually or use allowed objects if you need to store some data during a safe request (for example, logging some data for your analytics).

Allowed objects

There are rare cases which still justify persisting objects during safe requests. For example, your application might want to generate thumbnails of images during a GET request and persist the resulting PersistentResource instances.

For these cases it is possible to allow specific objects via the Persistence Manager:

```
$this->persistenceManager->allowObject($thumbnail);
$this->persistenceManager->allowObject($thumbnail->getResource());
```

Be very careful and think twice before using this method since many security measures are not active during "safe" request methods.

Dealing with big result sets

If the amount of the stored data increases, receiving all objects using a `findAll()` may consume a lot more memory than available. In this cases, you can use the `findAllIterator()`. This method returns an `IterableResult` over which you can iterate, getting only one object at a time:

```
$iterator = $this->postRepository->findAllIterator();
foreach ($this->postRepository->iterate($iterator) as $post) {
    // Iterate over all posts
}
```

Conventions for File and Class Names

To allow Flow to detect the object type a repository is responsible for, certain conventions need to be followed:

- Domain models should reside in a *Domain/Model* directory
- Repositories should reside in a *Domain/Repository* directory and be named `<ModelName>Repository`
- Aside from `Model` versus `Repository` the qualified class class names should be the same for corresponding classes
- Repositories must implement `\Neos\Flow\Persistence\RepositoryInterface` (which is already the case when extending `\Neos\Flow\Persistence\Repository` or `\Neos\Flow\Persistence\Doctrine\Repository`)

Example: Conventions for model and repository naming

```
\Neos
  \Blog
    \Domain
      \Model
        Blog
        Post
      \Repository
        BlogRepository
        PostRepository
```

Another way to bind a repository to a model is to define a class constant named `ENTITY_CLASSNAME` in your repository and give it the desired model name as value. This should be done only when following the conventions outlined above is not feasible.

Lazy Loading

Lazy Loading is a feature that can be equally helpful and dangerous when it comes to optimizing your application. Flow defaults to lazy loading when using Doctrine, i.e. it loads all the data in an object as soon as you fetch the object from the persistence layer but does not fetch data of associated objects. This avoids massive amounts of objects being reconstituted if you have a large object tree. Instead it defers property thawing in objects until the point when those properties are really needed.

The drawback of this: If you access associated objects, each access will fire a request to the persistent storage now. So there might be situations when eager loading comes in handy to avoid excessive database roundtrips. Eager loading is the default when using the *Generic* persistence mechanism and can be achieved for the Doctrine 2 ORM by using join operations in DQL or specifying the fetch mode in the mapping configuration.

Doctrine Persistence

Doctrine 2 ORM is used by default in Flow. Aside from very few internal changes it consists of the regular Doctrine ORM, DBAL, Migrations and Common libraries and is tied into Flow by some glue code and (most important) a custom annotation driver for metadata consumption.

Requirements and restrictions

There are some rules imposed by Doctrine (and/or Flow) you need to follow for your entities (and value objects). Most of them are good practice anyway, and thus are not really restrictions.

- Entity classes must not be `final` or contain `final` methods.
- Persistent properties of any entity class should always be `protected`, not `public`, otherwise lazy-loading might not work as expected.
- Implementing `__clone()` or `__wakeup()` is not a problem with Flow, as the instances always have an identity. If using your own identity properties, you must wrap any code you intend to run in those methods in an identity check.
- Entity classes in a class hierarchy that inherit directly or indirectly from one another must not have a mapped property with the same name.
- Entities cannot use `func_get_args()` to implement variable parameters. The proxies generated by Doctrine do not support this for performance reasons and your code might actually fail to work when violating this restriction.

Persisted instance variables must be accessed only from within the entity instance itself, not by clients of the entity. The state of the entity should be available to clients only through the entity's methods, i.e. getter/setter methods or other business methods.

Collection-valued persistent fields and properties must be defined in terms of the `Doctrine\Common\Collections\Collection` interface. The collection implementation type may be used by the application to initialize fields or properties before the entity is made persistent. Once the entity becomes managed (or detached), subsequent access must happen through the interface type.

Metadata mapping

The Doctrine 2 ORM needs to know a lot about your code to be able to persist it. Natively Doctrine 2 supports the use of annotations, XML, YAML and PHP to supply that information. In Flow, only annotations are supported, as this aligns with the philosophy behind the framework.

Annotations for the Doctrine Persistence

The following table lists the most common annotations used by the persistence framework with their name, scope and meaning:

Persistence-related code annotations

Annotation	Scope	Meaning
`Entity`	Class	Declares a class as an Entity.
`ValueObject`	Class	Declares a class as a Value Object, allowing the persistence framework to reuse an existing object if one exists.
`Column`	Variable	Allows to take influence on the column actually generated for this property in the database. Particularly useful with string properties to limit the space used or to enable storage of more than 255 characters.
`ManyToOne`, `OneToMany`, `ManyToMany`, `OneToOne`	Variable	Defines the type of object associations, refer to the Doctrine 2 documentation for details. The most obvious difference to plain Doctrine 2 is that the `targetEntity` parameter can be omitted, it is taken from the `@var` annotation. The `cascade` attribute is set to cascade all operations on associations within aggregate boundaries. In that case orphanRemoval is turned on as well.
`@var`	Variable	Is used to detect the type a variable has. For collections, the type is given in angle brackets.
`Transient`	Variable	Makes the persistence framework ignore the variable. Neither will it's value be persisted, nor will it be touched during reconstitution.
`Identity`	Variable	Marks the variable as being relevant for determining the identity of an object in the domain. For all class properties marked with this, a (compound) unique index will be created in the database.

Doctrine supports many more annotations, for a full reference please consult the Doctrine 2 ORM documentation.

On Value Object handling with Doctrine

Doctrine 2.5 supports value objects in the form of embeddable objects[5]. This means that the value object properties will directly be included in the parent entities table schema. However, Doctrine doesn't currently support embeddable collections[6]. Therefore, Flow supports two types of value objects: readonly entities and embedded

By default, Flow will use the readonly version, as that is more flexible and also works in collections. However, this comes with some architectural drawbacks, because the value object thereby is actually treated like an entity with an identifier, which contradicts the very definition of a value object.

The behaviour of non-embedded Value Objects is as follows:

- Value Objects are marked immutable as with the `ReadOnly` annotation of Doctrine.
- Each Value Object will internally be referenced by an identifier that is automatically generated from it's property values after construction.
- If the relation to a Value Object is annotated as OneTo* or ManyTo*, the Value Object will be persisted in it's own table. Otherwise, unless you override the type using `Column` Value Objects will be stored as serialized object in the database.
- Upon persisting Value Objects already present in the underlying database they will be deduplicated by being referenced through the identifier.

For cases where a *ToMany relation to a Value Object is not needed, the embedded form is the more natural way to persist value objects. You can therefore set the annotation property `embedded` to true, which will cause the Value Object to be embedded inside all Entities that reference it.

The behaviour of embedded Value Objects is as follows:

- Every entity having a property of type embedded Value Object will get all the properties of the Value Object included in it's schema.
- Unless you specify the `Embedded` Annotation on the relation property, the schema prefix will be the property name.

```
/**
 * @Flow\ValueObject(embedded=true)
 */
class ValueObject {
   ...
}

class SomeEntity {

       /**
        * @var ValueObject
        */
       protected $valueObject;
```

[5] https://doctrine-orm.readthedocs.org/en/latest/tutorials/embeddables.html
[6] https://github.com/doctrine/doctrine2/issues/3579

Custom Doctrine mapping types

Doctrine provides a way to develop custom mapping types as explained in the documentation ([#doctrineMapping-Types]).

Registration of those types in a Flow application is done through settings:

```
Neos:
  Flow:
    persistence:
      doctrine:
        # DBAL custom mapping types can be registered here
        dbal:
          mappingTypes:
            'mytype':
              dbType: 'db_mytype'
              className: 'Acme\Demo\Doctrine\DataTypes\MyType'
```

The custom type can then be used:

```
class SomeModel {

      /**
       * Some custom type property
       *
       * @ORM\Column(type="mytype")
       * @var string
       */
      protected $mytypeProperty;
```

On the Doctrine Event System

Doctrine provides a flexible event system to allow extensions to plug into different parts of the persistence. Therefore two methods to get notification of doctrine events are possible - through the EventSubscriber interface and registering EventListeners. Flow allows for easily registering both with Doctrine through the configuration settings `Neos.Flow.persistence.doctrine.eventSubscribers` and `Neos.Flow.persistence.doctrine.eventListeners` respectively. EventSubscribers need to implement the `Doctrine\Common\EventSubscriber` Interface and provide a list of the events they want to subscribe to. EventListeners need to be configured for the events they want to listen on, but do not need to implement any specific Interface. See the documentation ([7]) for more information on the Doctrine Event System.

Example: Configuration for Doctrine EventSubscribers and EventListeners:

```
Neos:
  Flow:
    persistence:
      doctrine:
        eventSubscribers:
          - 'Foo\Bar\Events\EventSubscriber'
        eventListeners:
          -
            events: ['onFlush', 'preFlush', 'postFlush']
            listener: 'Foo\Bar\Events\EventListener'
```

[7] https://doctrine-orm.readthedocs.org/en/latest/reference/events.html

On the Doctrine Filter System

Doctrine provides a filter system that allows developers to add SQL to the conditional clauses of queries, regardless the place where the SQL is generated (e.g. from a DQL query, or by loading).

Flow allows for easily registering Filters with Doctrine through the configuration setting `Neos.Flow.persistence.doctrine.filters`.

Example: Configuration for Doctrine Filters:

```
Neos:
  Flow:
    persistence:
      doctrine:
        filters:
          'my-filter-name': 'Acme\Demo\Filters\MyFilter'
```

See the Doctrine documentation ([8]) for more information on the Doctrine Filter System.

Note: If you create a filter and run into fatal errors caused by overriding a final `__construct()` method in one of the Doctrine classes, you need to add `@Flow\Proxy(false)` to your filter class to prevent Flow from building a proxy, which causes this error.

Warning: Custom SqlFilter implementations - watch out for data privacy issues!

If using custom SqlFilters, you have to be aware that the SQL filter is cached by doctrine, thus your SqlFilter might not be called as often as you might expect. This may lead to displaying data which is not normally visible to the user!

Basically you are not allowed to call *setParameter* inside *addFilterConstraint*; but setParameter must be called *before* the SQL query is actually executed. Currently, there's no standard Doctrine way to provide this; so you manually can receive the filter instance from *$entityManager->getFilters()->getEnabledFilters()* and call *setParameter()* then.

Alternatively, you can register a global context object in *Neos.Flow.aop.globalObjects* and use it to provide additional identifiers for the caching by letting these global objects implement *CacheAwareInterface*; effectively segregating the Doctrine cache some more.

Custom Doctrine DQL functions

Doctrine allows custom functions for use in DQL. In order to configure these for the use in Flow, use the following Settings:

```
Neos:
  Flow:
    persistence:
      doctrine:
        dql:
          customStringFunctions:
            'SOMEFUNCTION': 'Acme\Demo\Persistence\Ast\SomeFunction'
          customNumericFunctions:
```

(continues on next page)

[8] https://doctrine-orm.readthedocs.org/en/latest/reference/filters.html#filters

(continued from previous page)

```
            'FLOOR': 'Acme\Demo\Persistence\Ast\Floor'
            'CEIL': 'Acme\Demo\Persistence\Ast\Ceil'
          customDatetimeFunctions:
            'UTCDIFF': 'Acme\Demo\Persistence\Ast\UtcDiff'
```

See the Doctrine documentation ([2]) for more information on the Custom DQL functions.

Metadata and Query Cache

Flow automatically configures a cache for the Doctrine metadata, the used cache is the `Flow_Persistence_Doctrine` cache. The result cache is configured as well, the used cache is `Flow_Persistence_Doctrine_Results`.

This happens in `\Neos\Flow\Persistence\Doctrine\EntityManagerConfiguration::applyCacheConfigurat`

The use of the result cache can be enabled globally using the `Neos.Flow.persistence.cacheAllQueryResults` setting or on a per-query level by using the `$cacheResult` parameter of the `Query::execute()` method.

See https://www.doctrine-project.org/projects/doctrine-dbal/en/2.13/reference/caching.html for more information.

Using Doctrine's Second Level Cache

Doctrine provides a second level cache that further improves performance of relation queries beyond the result query cache.

See the Doctrine documentation ([3]) for more information on the second level cache. Flow allows you to enable and configure the second level cache through the configuration setting `Neos.Flow.persistence.doctrine.secondLevelCache`.

Example: Configuration for Doctrine second level cache:

```
Neos:
  Flow:
    persistence:
      doctrine:
        secondLevelCache:
          enable: true
          defaultLifetime: 3600
          regions:
            'my_entity_region': 7200
```

[2] http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html#adding-your-own-functions-to-the-dql-language
[3] https://www.doctrine-project.org/projects/doctrine-orm/en/2.10/reference/second-level-cache.html

Customizing Doctrine EntityManager

For any cases that are not covered with the above options, Flow provides two convenient signals to hook into the setup of the doctrine EntityManager.

The *beforeDoctrineEntityManagerCreation* signal provides you with the DBAL connection, the doctrine configuration and EventManager classes, that you can change before the actual EntityManager is instantiated.

The *afterDoctrineEntityManagerCreation* signal provides the doctrine configuration and EntityManager instance, in order to to further set options.

Note: All above configuration options through the settings are actually implemented as slots to the before mentioned signals. If you want to take some look how this works, check the *NeosFlowPersistenceDoctrineEntityManagerConfiguration* class.

Differences between Flow and plain Doctrine

The custom annotation driver used by Flow to collect mapping information from the code makes a number of things easier, compared to plain Doctrine 2.

`Entity` `repositoryClass` can be left out, if you follow the naming rules for your repository classes explained above.

`Table` `name` does not default to the unqualified entity classname, but a name is generated from class name, package key and more elements to make it unique.

`Id` Can be left out, as it is automatically generated, this means you also do not need `@GeneratedValue`. Every entity will get a property injected that is filled with an UUID upon instantiation and used as technical identifier.

If an `@Id` annotation is found, it is of course used as is and no magic will happen.

`Column` Can usually be left out altogether, as the vital *type* information can be read from the `@var` annotation on a class member.

Important: Since PHP does not differentiate between short and long strings, but databases do, you must use `@Column(type="text")` if you intend to store more than 255 characters in a string property.

`OneToOne`, `OneToMany`, `ManyToOne`, `ManyToMany` `targetEntity` can be omitted, it is read from the `@var` annotation on the property. Relations to Value Objects will be `cascade` `persist` by default and relations to non aggregate root entities will be `cascade` `all` by default.

`JoinTable`, `JoinColumn` Can usually be left out completely, the needed information is gathered automatically But *when using a self-referencing association*, you will need to help Flow a little, so it doesn't generate a join table with only one column.

Example: JoinTable annotation for a self-referencing annotation

```
/**
 * @var \Doctrine\Common\Collections\ArrayCollection<\Neos\Blog\Domain\Model\Post>
 * @ORM\ManyToMany
 * @ORM\JoinTable(inverseJoinColumns={@ORM\JoinColumn(name="related_id")})
 */
protected $relatedPosts;
```

Without this, the created table would not contain two columns but only one, named after the identifiers of the associated entities - which is the same in this case.

`DiscriminatorColumn`, `DiscriminatorMap` Can be left out, as they are automatically generated.

The generation of this metadata is slightly more expensive compared to the plain Doctrine `AnnotationDriver`, but since this information can be cached after being generated once, we feel the gain when developing outweighs this easily.

Tip: Anything you explicitly specify in annotations regarding Doctrine, has precedence over the automatically generated metadata. This can be used to fully customize the mapping of database tables to models.

Here is an example to illustrate the things you can omit, due to the automatisms in the Flow annotation driver.

Example: Annotation equivalents in Flow and plain Doctrine 2

An entity with only the annotations needed in Flow:

```
/**
 * @Flow\Entity
 */
class Post {

  /**
   * @var \Neos\Blog\Domain\Model\Blog
   * @ORM\ManyToOne(inversedBy="posts")
   */
  protected $blog;

  /**
   * @var string
   * @ORM\Column(length=100)
   */
  protected $title;

  /**
   * @var \DateTime
   */
  protected $date;

  /**
   * @var string
   * @ORM\Column(type="text")
   */
  protected $content;

  /**
   * @var \Doctrine\Common\Collections\ArrayCollection<\Neos\Blog\Domain\Model\
↪Comment>
   * @ORM\OneToMany(mappedBy="post")
   * @ORM\OrderBy({"date" = "DESC"})
   */
  protected $comments;
```

The same code with all annotations needed in plain Doctrine 2 to result in the same metadata:

```
/**
 * @ORM\Entity(repositoryClass="Neos\Blog\Domain\Model\Repository\PostRepository")
```

(continues on next page)

(continued from previous page)

```
 * @ORM\Table(name="blog_post")
 */
class Post {

  /**
   * @var string
   * @ORM\Id
   * @ORM\Column(name="persistence_object_identifier", type="string", length=40)
   */
  protected $Persistence_Object_Identifier;

  /**
   * @var \Neos\Blog\Domain\Model\Blog
   * @ORM\ManyToOne(targetEntity="Neos\Blog\Domain\Model\Blog", inversedBy="posts")
   * @ORM\JoinColumn(name="blog_blog", referencedColumnName="persistence_object_
↪identifier")
   */
  protected $blog;

  /**
   * @var string
   * @ORM\Column(type="string", length=100)
   */
  protected $title;

  /**
   * @var \DateTime
   * @ORM\Column(type="datetime")
   */
  protected $date;

  /**
   * @var string
   * @ORM\Column(type="text")
   */
  protected $content;

  /**
   * @var \Doctrine\Common\Collections\ArrayCollection<\Neos\Blog\Domain\Model\
↪Comment>
   * @ORM\OneToMany(targetEntity="Neos\Blog\Domain\Model\Comment", mappedBy="post",
    cascade={"all"}, orphanRemoval=true)
   * @ORM\OrderBy({"date" = "DESC"})
   */
  protected $comments;
```

Schema management

Doctrine offers a *Migrations* system as an add-on part of its DBAL for versioning of database schemas and easy deployment of changes to them. There exist a number of commands in the Flow CLI toolchain to create and deploy migrations.

A Migration is a set of commands that bring the schema from one version to the next. In the simplest form that means creating a new table, but it can be as complex as renaming a column and converting data from one format to another along the way. Migrations can also be reversed, so one can migrate up and down.

Each Migration is represented by a PHP class that contains the needed commands. Those classes come with the package they relate to, they have a name that is based on the time they were created. This allows correct ordering of migrations coming from different packages.

Query the schema status

To learn about the current schema and migration status, run the following command:

```
$ ./flow flow:doctrine:migrationstatus
```

This will produce output similar to the following, obviously varying depending on the actual state of schema and active packages:

Example: Migration status report

```
== Configuration
   >> Name:                                               Doctrine Database Migrations
   >> Database Driver:                                    pdo_mysql
   >> Database Name:                                      flow
   >> Configuration Source:                               manually configured
   >> Version Table Name:                                 flow_doctrine_
↪migrationstatus
   >> Migrations Namespace:                               Neos\Flow\Persistence\
↪Doctrine\Migrations
   >> Migrations Target Directory:                        /path/to/Data/
↪DoctrineMigrations
   >> Current Version:                                    0
   >> Latest Version:                                     2011-06-13 22:38:37␣
↪(20110613223837)
   >> Executed Migrations:                                0
   >> Available Migrations:                               1
   >> New Migrations:                                     1

== Migration Versions
   >> 2011-06-13 22:38:37 (20110613223837)                not migrated
```

Whenever a version number needs to be given to a command, use the short form as shown in parentheses in the output above. The migrations directory in the output is only used when creating migrations, see below for details on that.

Deploying migrations

On a pristine database it is very easy to create the tables needed with the following command:

```
$ ./flow flow:doctrine:migrate
```

This will result in output that looks similar to the following:

```
Migrating up to 20110613223837 from 0

  ++ migrating 20110613223837

     -> CREATE TABLE flow_resource_resourcepointer (hash VARCHAR(255) NOT NULL,␣
↪PRIMARY KEY(hash)) ENGINE = InnoDB
     -> ALTER TABLE flow_resource_resource ADD FOREIGN KEY (flow_resource_
↪resourcepointer) REFERENCES flow_resource_resourcepointer(hash)

  ++ migrated (1.31s)

  ------------------------

  ++ finished in 1.31
  ++ 1 migrations executed
  ++ 6 sql queries
```

This will deploy all migrations delivered with the currently active packages to the configured database. During that process it will display all the SQL statements executed and a summary of the deployed migrations at the and. You can do a dry run using:

```
$ ./flow flow:doctrine:migrate --dry-run
```

This will result in output that looks similar to the following:

```
Executing dry run of migration up to 20110613223837 from 0

  ++ migrating 20110613223837

     -> CREATE TABLE flow_resource_resourcepointer (hash VARCHAR(255) NOT NULL,␣
↪PRIMARY KEY(hash)) ENGINE = InnoDB
     -> ALTER TABLE flow_resource_resource ADD FOREIGN KEY (flow_resource_
↪resourcepointer) REFERENCES flow_resource_resourcepointer(hash)

  ++ migrated (0.09s)

  ------------------------

  ++ finished in 0.09
  ++ 1 migrations executed
  ++ 6 sql queries
```

to see the same output but without any changes actually being done to the database. If you want to inspect and possibly adjust the statements that would be run and deploy manually, you can write to a file:

```
$ ./flow flow:doctrine:migrate --path <where/to/write/the.sql>
```

This will result in output that looks similar to the following:

```
Writing migration file to "<where/to/write/the.sql>"
```

Important: When actually making manual changes, you need to keep the `flow_doctrine_migrationstatus` table updated as well! This is done with the `flow:doctrine:migrationversion` command. It takes a `--version` option together with either an `--add` or `--delete` flag to add or remove the given version in the `flow_doctrine_migrationstatus` table. It does not execute any migration code but simply marks the given version as migrated or not.

Reverting migrations

The migrate command takes an optional `--version` option. If given, migrations will be executed up or down to reach that version. This can be used to revert changes, even completely:

```
$ ./flow flow:doctrine:migrate --version <version> --dry-run
```

This will result in output that looks similar to the following:

```
Executing dry run of migration down to 0 from 20110613223837

  -- reverting 20110613223837

     -> ALTER TABLE flow_resource_resource DROP FOREIGN KEY
     -> DROP TABLE flow_resource_resourcepointer
     -> DROP TABLE flow_resource_resource
     -> DROP TABLE flow_security_account
     -> DROP TABLE flow_resource_securitypublishingconfiguration
     -> DROP TABLE flow_policy_role

  -- reverted (0.05s)

  ------------------------

  ++ finished in 0.05
  ++ 1 migrations executed
  ++ 6 sql queries
```

Executing or reverting a specific migration

Sometimes you need to deploy or revert a specific migration, this is possible as well.

```
$ ./flow flow:doctrine:migrationexecute --version <20110613223837> --direction
↪<direction> --dry-run
```

This will result in output that looks similar to the following:

```
-- reverting 20110613223837

   -> ALTER TABLE flow_resource_resource DROP FOREIGN KEY
   -> DROP TABLE flow_resource_resourcepointer
   -> DROP TABLE flow_resource_resource
   -> DROP TABLE flow_security_account
```

(continues on next page)

(continued from previous page)

```
   -> DROP TABLE flow_resource_securitypublishingconfiguration
   -> DROP TABLE flow_policy_role

-- reverted (0.41s)
```

As you can see you need to specify the migration `--version` you want to execute. If you want to revert a migration, you need to give the `--direction` as shown above, the default is to migrate "up". The `--dry-run` and and `--output` options work as with `flow:doctrine:migrate`.

Creating migrations

Migrations make the schema match when a model changes, but how are migrations created? The basics are simple, but rest assured that database details and certain other things make sure you'll need to practice... The command to scaffold a migration is the following:

```
$ ./flow flow:doctrine:migrationgenerate
```

This will result in output that looks similar to the following:

```
Generated new migration class!

Do you want to move the migration to one of these packages?
  [0 ] Don't Move
  [1 ] Neos.Diff
  [2 ] ...
```

You should pick the package that your new migration covers, it will then be moved as requested. The command will output the path to generated migration and suggest some next steps to take.

Important: If you decide not to move the file, it will be put into *Data/DoctrineMigrations/*.

That directory is only used when creating migrations. The migrations visible to the system are read from *Migrations/<DbPlatForm>* in each package. The *<DbPlatform>* represents the target platform, e.g. `Mysql` (as in Doctrine DBAL but with the first character uppercased).

Looking into that file reveals a basic migration class already filled with the differences detected between the current schema and the current models in the system:

Example: Migration generated based on schema/model differences

```
namespace Neos\Flow\Persistence\Doctrine\Migrations;

use Doctrine\DBAL\Migrations\AbstractMigration,
  Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your need!
 */
class Version20110624143847 extends AbstractMigration {

  /**
   * @param Schema $schema
   * @return void
   */
```

(continues on next page)

(continued from previous page)

```
  public function up(Schema $schema) {
      // this up() migration is autogenerated, please modify it to your needs
    $this->abortIf($this->connection->getDatabasePlatform()->getName() != "mysql");

    $this->addSql("CREATE TABLE party_abstractparty (...) ENGINE = InnoDB");
  }

  /**
   * @param Schema $schema
   * @return void
   */
  public function down(Schema $schema) {
      // this down() migration is autogenerated, please modify it to your needs
    $this->abortIf($this->connection->getDatabasePlatform()->getName() != "mysql");

    $this->addSql("DROP TABLE party_abstractparty");
  }
}
```

To create an empty migration skeleton, pass `--diff-against-current 0` to the command.

After you generated a migration, you will probably need to clean up a little, as there might be differences being picked up that are not useful or can be optimized. An example is when you rename a model: The migration will drop the old table and create the new one, but what you want instead is to *rename* the table. Also you must to make sure each finished migration file only deals with one package and then move it to the *Migrations* directory in that package. This way different packages can be mixed and still a reasonable migration history can be built up.

Ignoring tables

For tables that are not known to the schema because they are code-generated or come from a different system sharing the same database, the `flow:doctrine:migrationgenerate` command will generate corresponding `DROP TABLE` statements. In this case you can use the `--filter-expression` flag to generate migrations only for tables matching the given pattern:

```
$ ./flow flow:doctrine:migrationgenerate --filter-expression '^your_package_.*'
```

Will only affect tables starting with "your_package_".

To permanently skip certain tables the `ignoredTables` setting can be used:

```
Neos:
  Flow:
    persistence:
      doctrine:
        migrations:
          ignoredTables:
            'autogenerated_.*': TRUE
            'wp_.*: TRUE
```

Will ignore table starting with "autogenerated_" or "wp_" by default (the *–filter-expression* flag overrules this setting).

Schema updates without migrations

Migrations are the recommended and preferred way to bring your schema up to date. But there might be situations where their use is not possible (e.g. no migrations are available yet for the RDBMS you are using) or not wanted (because of, um... something). The there are two simple commands you can use to create and update your schema.

To create the needed tables you can call `./flow flow:doctrine:create` and it will create all needed tables. If any target table already exists, an error will be the result.

To update an existing schema to match with the current mapping metadata (i.e. the current model structure), use `./flow flow:doctrine:update` to have missing items (fields, indexes, ...) added. There is a flag to disable the safe mode used by default. In safe mode, Doctrine tries to keep existing data as far as possible, avoiding lossy actions.

Warning: Be careful, the update command might destroy data, as it could drop tables and fields irreversibly. It also doesn't respect the `ignoredTables` settings (see previous section).

Both commands also support `--output <write/here/the.sql>` to write the SQL statements to the given file instead of executing it.

Tip: If you created or updated the schema this way, you should afterwards execute `flow:doctrine:migrationversion --version all --add` to avoid migration errors later.

Doctrine Connection Wrappers - Primary/Replica Connections

Doctrine 2 allows to create Connection wrapper classes, that change the way Doctrine connects to your database. A common use case is a primary/replica setup, with one primary server and several read replicas that share the load for all reading queries. Doctrine already provides a wrapper for such a connection and you can configure Flow to use that connection wrapper by setting the following options in your packages `Settings.yaml`:

```
Neos:
  Flow:
    persistence:
      backendOptions:
        wrapperClass: 'Doctrine\DBAL\Connections\PrimaryReadReplicaConnection'
        primary:
          host: '127.0.0.1'      # adjust to your master database host
          dbname: 'master'       # adjust to your database name
          user: 'user'           # adjust to your database user
          password: 'pass'       # adjust to your database password
        replicas:
          replica1:
            host: '127.0.0.1'        # adjust to your slave database host
            dbname: 'replica1'       # adjust to your database name
            user: 'user'             # adjust to your database user
            password: 'pass'         # adjust to your database password
```

Note: In doctrine/dbal versions lower then 2.11 the wrapper class was named *MasterSlaveConnection*, so you need to adjust to that if you are such a version.

With this setup, Doctrine will use one of the replica connections picked once per request randomly for all queries until the first writing query (e.g. insert or update) is executed. From that point on the primary server will be used solely.

This is to solve the problems of replication lag and possibly inconsistent query results.

Tip: You can also setup the primary database as a replica, if you want to also use it for load-balancing reading queries. However, this might lead to higher load on the primary database and should be well observed.

Known issues

- When using PostgreSQL the use of the `object`, and `array` mapping types is not possible, this is caused by Doctrine using `serialize()` to prepare data that is stored in text column (contained zero bytes truncate the string and lead to error during hydration).[9]

  The Flow mapping types `flow_json_array` and `objectarray` provide solutions for this.

- When using PostgreSQL the use of the `json_array` mapping type can lead to issues when queries need comparisons on such columns (e.g. when grouping or doing distinct queries), because the `json` type used by Doctrine doesn't support comparisons.

  The Flow mapping type `flow_json_array` uses the `jsonb` type available as of PostgreSQL 9.4, circumventing this restriction.

Generic Persistence

What is now called *Generic* Persistence, used to be the only persistence layer in Flow. Back in those days there was no ORM available that fit our needs. That being said, with the advent of Doctrine 2, your best bet as a PHP developer is to use that instead of any home-brewn ORM.

Note: The *Generic* persistence is deprecated as of Flow 6.0 and will be dropped in Flow 7.0.

When your target is not a relational database, things look slightly different, which is why the "old" code is still available for use, primarily by alternative backends like the ones for CouchDB or Solr, that are available. Using the Generic persistence layer to target a RDBMS is still possible, but probably only useful for rare edge cases.

Switching to Generic Persistence

To switch to Generic persistence you need to configure Flow like this.

Objects.yaml:

```
Neos\Flow\Persistence\PersistenceManagerInterface:
  className: 'Neos\Flow\Persistence\Generic\PersistenceManager'

Neos\Flow\Persistence\QueryResultInterface:
  scope: prototype
  className: 'Neos\Flow\Persistence\Generic\QueryResult'
```

Settings.yaml:

[9] http://www.doctrine-project.org/jira/browse/DDC-3241

```
Flow:
  persistence:
    doctrine:
      enable: FALSE
```

When installing generic backend packages, like CouchDB, the needed object configuration should be contained in them, for the connection settings, consult the package's documentation.

Metadata mapping

The persistence layer needs to know a lot about your code to be able to persist it. In Flow, the needed data is given in the source code through annotations, as this aligns with the philosophy behind the framework.

Annotations for the Generic Persistence

The following table lists all annotations used by the persistence framework with their name, scope and meaning:

Persistence-related code annotations

Annotation	Scope	Meaning
`Entity`	Class	Declares a class as an Entity.
`ValueObject`	Class	Declares a class as a Value Object, allowing the persistence framework to reuse an existing object if one exists.
`@var`	Variable	Is used to detect the type a variable has.
`Transient`	Variable	Makes the persistence framework ignore the variable. Neither will it's value be persisted, nor will it be touched during reconstitution.
`Identity`	Variable	Marks the variable as being relevant for determining the identity of an object in the domain.
`Lazy`	Class, Variable	When reconstituting the value of this property will be loaded only when the property is used. Note: This is only supported for properties of type `\SplObjectStorage` and objects (marked with `Lazy` in their source code, see below).

Enabling Lazy Loading

If a class should be able to be lazy loaded by the PDO backend, you need to annotate it with `@lazy` in the class level docblock. This is done to avoid creating proxy classes for objects that should never be lazy loaded anyway. As soon as that annotation is found, AOP is used to weave lazy loading support into your code that intercepts all method calls and initializes the object before calling the expected method. Such a proxy class is a subclass of your class, as such it work fine with type hinting and checks and can be used the same way as the original class.

To actually mark a property for lazy loading, you need to add the `@lazy` annotation to the property docblock in your code. Then the persistence layer will skip loading the data for that object and the object properties will be thawed when the object is actually used.

How @lazy annotations interact

Class	Property	Effect
`Lazy`	`Lazy`	The class' instances will be lazy loadable, and properties of that type will be populated with a lazy loading proxy.
`Lazy`	*none*	The class' instances will be lazy loadable, but that possibility will not be used.
none	`Lazy`	`\SplObjectStorage` will be reconstituted as a lazy loading proxy, for other types nothing happens. Properties of type `\SplObjectStorage` can always be lazy-loaded by adding the `Lazy` annotation on the property only. How and if lazy-loading is handled by alternative backends is up to the implementation.

Schema management

Whether other backends implement automatic schema management is up to the developers, consult the documentation of the relevant backend for details.

Inside the Generic Persistence

To the domain code the persistence handling transparent, aside from the need to add a few annotations. The custom repositories are a little closer to the inner workings of the framework, but still the inner workings are very invisible. This is how it is supposed to be, but a little understanding of how persistence works internally can help understand problems and develop more efficient client code.

Persisting a Domain Object

After an object has been added to a repository it will be seen when Flow calls `persistAll()` at the end of a script run. Internally all instances implementing the `\Neos\Flow\Persistence\RepositoryInterface` will be fetched and asked for the objects they hold. Those will then be handed to the persistence backend in use and processed by it.

Flow defines interfaces for persistence backends and queries, the details of how objects are persisted and queried are up to the persistence backend implementation. Have a look at the documentation of the respective package for more information. The following diagram shows (most of) the way an object takes from creation until it is persisted when using the suggested process:

Keep in mind that the diagram omits some details like dirty checking on objects and how exactly objects and their properties are stored.

Querying the Storage Backend

As we saw in the introductory example there is a query mechanism available that provides easy fetching of objects through the persistence framework. You ask for instances of a specific class that match certain filters and get back an array of those reconstituted objects. Here is a diagram of the internal process when using the suggested process:

For the developer the complexity is hidden between the query's `execute()` method and the array of objects being returned.

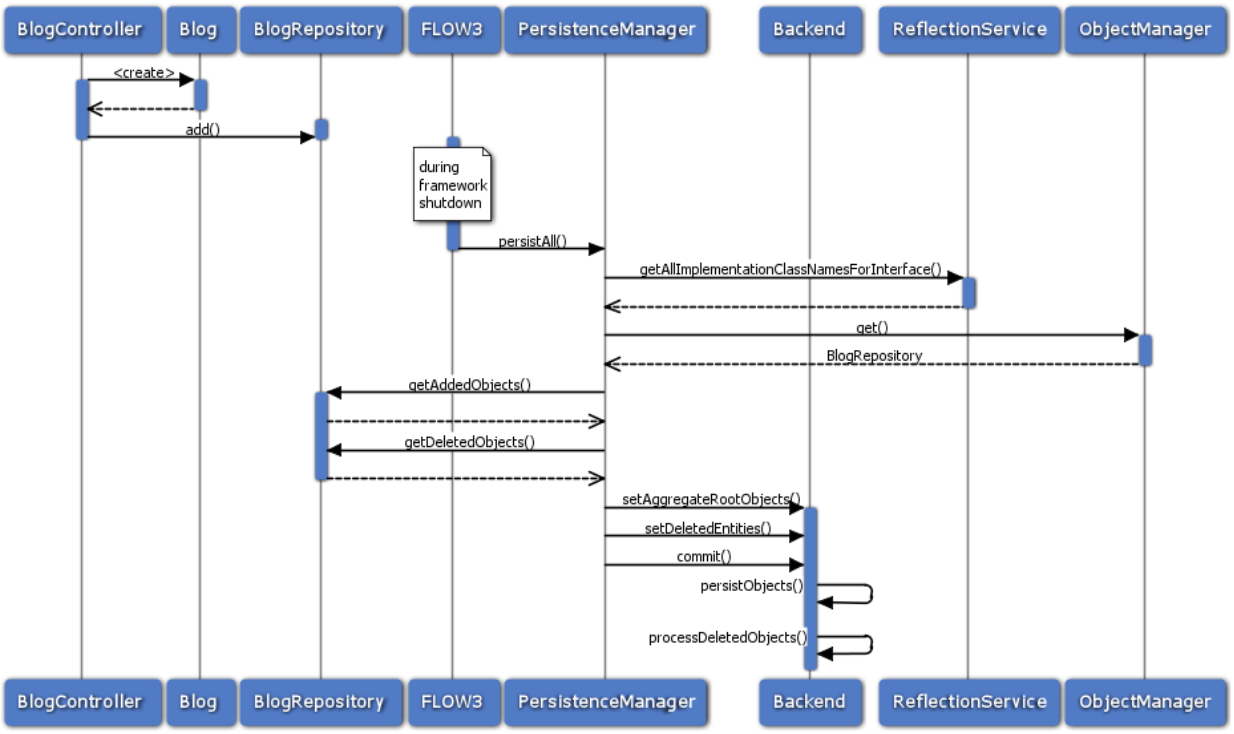

Fig. 25: Object persistence process

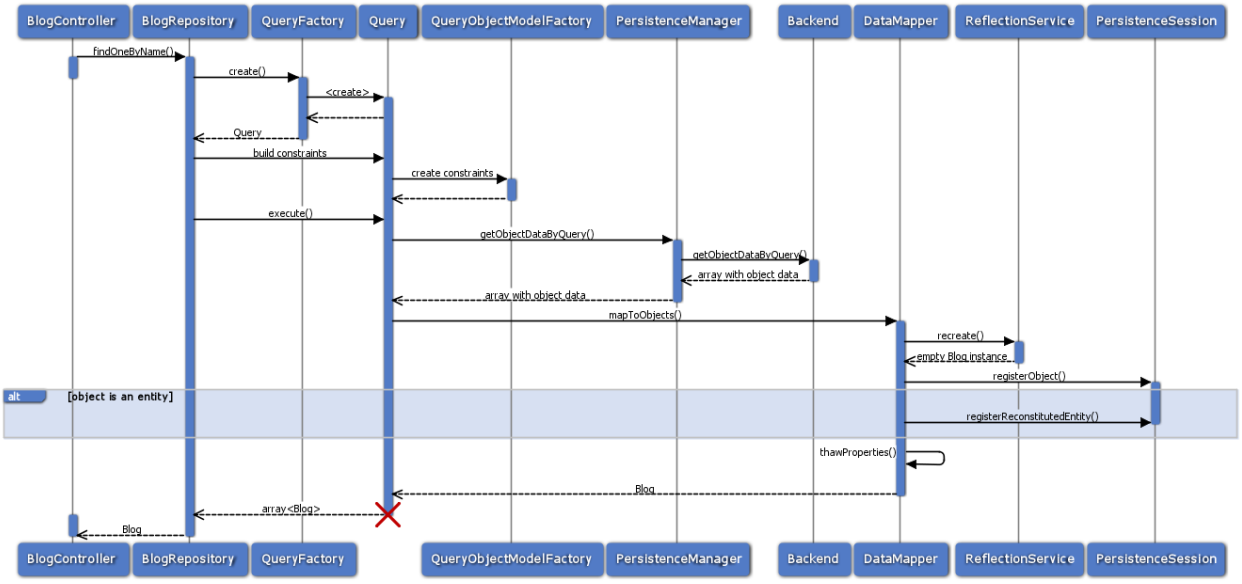

Fig. 26: Object querying and reconstitution process

2.3.7 HTTP Foundation

Most applications which are based on Flow are web applications. As the HTTP protocol is the foundation of the World Wide Web, it also plays an important role in the architecture of the Flow framework.

This chapter describes the mechanics behind Flow's request-response model, how it relates to the Model View Controller framework and which API functions you can use to deal with specific aspects of the HTTP request and response.

The HTTP 1.1 Specification

Although most people using or even developing for the web are aware of the fact that the Hypertext Transfer Protocol is responsible for carrying data around, considerably few of them have truly concerned themselves with the HTTP 1.1 specification.

The specification, RFC 2616, has been published in 1999 already but it is relevant today more than ever. If you've never fully read it, we recommend that you do so. Although it is a long read, it is important to understand the intentions and rules of the protocol before you can send cache headers or response codes in good conscience, or even claim that you developed a true REST service.

Application Flow

The basic walk through a Flow-based web application is as follows:

- the browser sends an HTTP request to a webserver
- the webserver calls Web/index.php and passes control over to Flow
- the BOOTSTRAP (\Neos\Flow\Core\Bootstrap) initializes the bare minimum and passes control to a suitable request handler
- by default, the HTTP REQUEST HANDLER (\Neos\Flow\Http\RequestHandler) takes over and runs a boot sequence which initializes all important parts of Flow
- the HTTP Request Handler builds an PSR-7 HTTP Request and Response object. The REQUEST OBJECT (\Psr\Http\Message\ServerRequestInterface) contains all important properties of the real HTTP request. The RESPONSE OBJECT (\PsrHttp\Message\ResponseInterface) in turn is empty and will be filled with information by a controller at a later point Both are stored in the so-called COMPONENTCONTEXT (\Neos\Flow\Http\Component\ComponentContext), which you need to use to access and/or replace any of the two.
- the HTTP Request Handler initializes the HTTP MIDDLEWARES CHAIN (\Neos\Flow\Http\Middleware\MiddlewaresChain), which is a PSR-15 RequestHandler implementation wrapping a configurable list of PSR-15 Middlewares. Currently this Middlewares chain only consists of a middleware that invokes the deprecated HTTP COMPONENT CHAIN (\Neos\Flow\Http\Component\ComponentChain), a set of independent units that have access to the current HTTP request and response and can share information amongst each other. The Component chain is fully configurable, but by default it consists of the following steps:
- the `trusted proxies` component verifies headers that override request information, like the host, port or client IP address to come from a server (reverse proxy) who's IP address is safe-listed in the settings.
- the `session cookie` component, which restores the session from a cookie and later sets the session cookie in the response.
- the `routing` component invokes the ROUTER (\Neos\Flow\Mvc\Routing\Router) to determine which controller and action is responsible for processing the request. This information (controller name, action name, arguments) is stored in the `ComponentContext`

- the `dispatching` component tries to invoke the corresponding controller action via the DISPATCHER (Neos\Flow\Mvc\Dispatcher)
- the controller, usually an ACTION CONTROLLER (\Neos\Flow\Mvc\Controller\ActionController), processes the request and modifies the given HTTP Response object which will, in the end, contain the content to display (body) as well as any headers to be passed back to the client
- the `standardsCompliance` component tries to make the HTTP Response standards compliant by adding required HTTP headers and setting the correct status code (if not already the case)
- Finally the RequestHandler sends the HTTP Response back to the browser

In practice, there are a few more intermediate steps being carried out, but in essence, this is the path a request is taking.

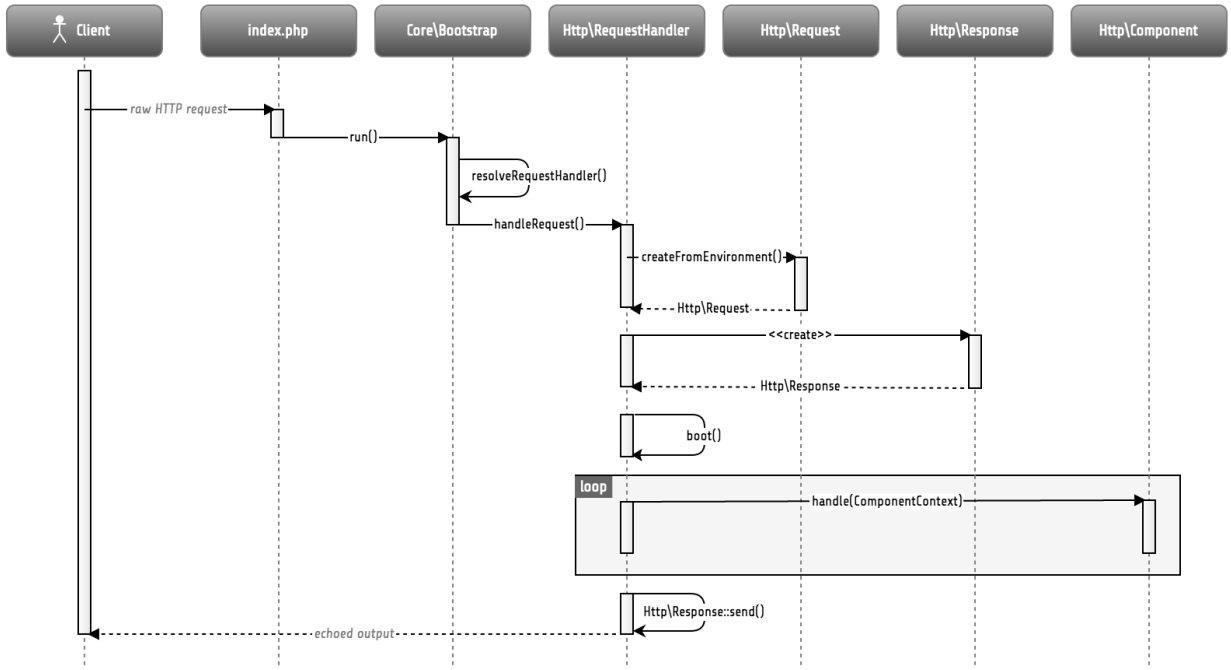

Fig. 27: Simplified application flow

The Response is modified within the HTTP Middlewares/Component Chain, visualized by the highlighted "loop" block above. The component chain is configurable. If no components were registered every request would result in a blank HTTP Response. The component chain is a component too, so chains can be nested. By default the base component chain is divided into three sub chains "preprocess", "process" and "postprocess". The "preprocess" chain is empty by default, the "process" chain contains components for "routing" and "dispatching" and the "postprocess" chain contains a "standardsCompliance" component:

The next sections shed some light on the most important actors of this application flow.

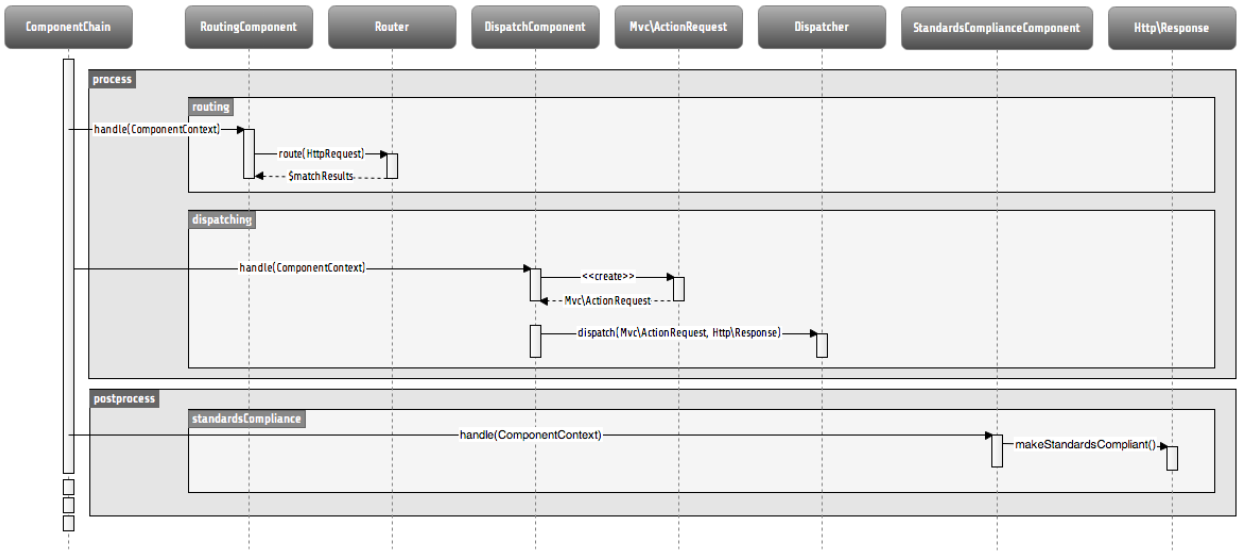

Fig. 28: Default HTTP Component Chain

Request Handler

The request handler is responsible for taking a request and responding in a manner the client understands. The default HTTP Request Handler invokes the `Bootstrap runtime sequence` and initializes the `HTTP Middlewares chain`. Other request handlers may choose a completely different way to handle requests. Although Flow also supports other types of requests (most notably, from the command line interface), this chapter only deals with HTTP requests.

Flow comes with a very slim bootstrap, which results in few code being executed before control is handed over to the request handler. This pays off in situations where a specialized request handler is supposed to handle specific requests in a very effective way. In fact, the request handler is responsible for executing big parts of the initialization procedures and thus can optimize the boot process by choosing only the parts it actually needs.

A request handler must implement the REQUESTHANDLER INTERFACE (\Neos\Flow\Core\RequestHandlerInterface) interface which, among others, contains the following methods:

```
public function handleRequest();

public function canHandleRequest();

public function getPriority();
```

On trying to find a suitable request handler, the bootstrap asks each registered request handler if it can handle the current request using `canHandleRequest()` – and if it can, how eager it is to do so through `getPriority()`. Request handlers responding with a high number as their priority, are preferred over request handlers reporting a lower priority. Once the bootstrap has identified a matching request handler, it passes control to it by calling its `handleRequest()` method.

Request handlers must first be registered in order to be considered during the resolving phase. Registration is done in the `Package` class of the package containing the request handler:

```
class Package extends BasePackage {

        public function boot(\Neos\Flow\Core\Bootstrap $bootstrap) {
                $bootstrap->registerRequestHandler(new \Acme\Foo\BarRequestHandler(
↪$bootstrap));
```

(continues on next page)

(continued from previous page)

```
        }

}
```

Middlewares Chain

Instead of registering a new RequestHandler the application workflow can also be altered by a custom `PSR-15 Middleware`. A HTTP middleware must implement the MIDDLEWARE INTERFACE (\Psr\Http\Server\MiddlewareInterface) that defines the `process($request, $next)` method:

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Server\MiddlewareInterface;
use Psr\Http\Server\RequestHandlerInterface;

/**
 * A sample HTTP middleware that intercepts the default handling and returns "bar" if␣
↪the request contains an argument "foo"
 */
class SomeMiddleware implements MiddlewareInterface {

  /**
   * @param ServerRequestInterface $httpRequest
   * @param RequestHandlerInterface $next
   * @return ResponseInterface
   */
  public function process(ServerRequestInterface $httpRequest,␣
↪RequestHandlerInterface $next): ResponseInterface;
    if (!$httpRequest->hasArgument('foo')) {
      // You may also return a new HttpResponse here and thereby short-cut the␣
↪further handling
      return $next->handle($httpRequest);
    }
    $httpResponse = $next->handle($httpRequest);
    return $httpResponse->withContent('bar');
  }

}
```

To activate a middleware, it must be configured in the `Settings.yaml`:

```
Neos:
  Flow:
    http:
      middlewares:
        'custom':
          position: 'before dispatch'
          middleware: 'Some\Package\Http\SomeHttpMiddleware'
```

With the `position` directive the order of a middleware within the chain can be defined. In this case the new component will be handled before the dispatch middleware that is configured in the Neos.Flow package. Note though, that any middleware will always be able to act on the request, so *before* any following middleware and also on the response, hence *after* the following middleware. A middleware chain basically works like a onion ring, where each middleware is a single layer of the onion around the inner core of the application. Each request passes inside through the layer and a response passes outside through the layer.

Component Chain

..note:

```
The Component Chain is considered deprecated as of Flow 6.3 and will be removed in a␣
↪later version. All components will
be replaced by `PSR-15 Middlewares`_ and an easy upgrade-path will be provided.
```

Instead of registering a new RequestHandler the application workflow can also be altered by a custom `HTTP Component`. A HTTP component must implement the COMPONENT INTERFACE (\Neos\Flow\Http\Component\ComponentInterface) that defines the `handle()` method:

```
  use Neos\Flow\Http\Component\ComponentInterface;
  use Neos\Flow\Http\Component\ComponentContext;

  /**
   * A sample HTTP component that intercepts the default handling and returns "bar"␣
↪if the request contains an argument "foo"
   */
  class SomeHttpComponent implements ComponentInterface {

          /**
           * @var array
           */
          protected $options;

          /**
           * @param array $options
           */
          public function __construct(array $options = array()) {
                  $this->options = $options;
          }

          /**
           * @param ComponentContext $componentContext
           * @return void
           */
          public function handle(ComponentContext $componentContext) {
                  $httpRequest = $componentContext->getHttpRequest();
                  if (!$httpRequest->hasArgument('foo')) {
                          return;
                  }
                  $httpResponse = $componentContext->getHttpResponse();
$modifiedResponse = $httpResponse->withContent('bar');
                  $componentContext->replaceHttpResponse($modifiedResponse);
          }
  }
```

The `ComponentContext` contains a reference to the current HTTP request and response, besides it can be used to pass arbitrary parameters to successive components. To activate a component, it must be configured in the `Settings.yaml`:

```
Neos:
  Flow:
    http:
      chain:
        'process':
```

(continues on next page)

(continued from previous page)

```
        chain:
          'custom':
            position: 'before routing'
            component: 'Some\Package\Http\SomeHttpComponent'
            componentOptions:
              'someOption': 'someValue'
```

With the `position` directive the order of a component within the chain can be defined. In this case the new component will be handled before the routing component that is configured in the Neos.Flow package. `componentOptions` is an optional key/value array with options that will be passed to the component's constructor.

Interrupting the chain

Sometimes it is necessary to stop processing of a chain in order to prevent successive components to be executed. For example if one wants to handle an AJAX request and prevent the default dispatching. This can be done by setting the `cancel` parameter of the `ComponentChain`:

```
/**
 * @param ComponentContext $componentContext
 * @return void
 */
public function handle(ComponentContext $componentContext) {
        // check if the request should be handled and return otherwise

        $componentContext->setParameter(\Neos\Flow\Http\Component\
↪ComponentChain::class, 'cancel', TRUE);
}
```

Note that component chains can be nested. By default the three sub chains `preprocess`, `process` and `postprocess` are configured. Setting the `cancel` parameter only affects the currently processed chain. With the examples from above the new component is added to the `process` chain. This way the `postprocess` chain is still handled even if the new component cancels the current chain.

Request

In the PSR-7 specification, a distinction is made between two different types of requests - incoming (`ServerRequest`) and outgoing (`Request`). Whenever you want to make an outgoing request, you can easily use the Guzzle `Request` class constructor for example with the respective arguments for method, uri, etc. and then pass that to e.g. a PSR-18 Http Client implementation. On the other side the incoming request is something you should never try to create an instance of yourself, as it is provided by the framework. In theory, you could also call the `ServerRequestFactory::createServerRequest` or the Guzzle `ServerRequest::fromGlobals()` convenience method, but this does not have any relation to the current request object handled by the framework. It will not have any of the processing from components applied and might therefore lead to unexpected results, like the trusted proxy headers `X-Forwarded-*` not being applied and the `ServerRequest` providing wrong protocol, host or client IP address. If you need access to the **current** HTTP `Request`, either create a Http Component or only access it inside the controller through the `ActionRequest` for inspecting:

```
public function myAction() {
        $requestBody = $this->request->getHttpRequest()->getParsedBody();
        ...
}
```

Creating an ActionRequest

Normally, you should not need to create an `ActionRequest` yourself. It only has meaning inside the `MVC` layer of the framework and is created before invoking the MVC dispatcher. If you do need to create an `ActionRequest` yourself to dispatch, such a request is always bound to an HTTP `ServerRequest`:

```
use Neos\Flow\Core\Bootstrap;
use Neos\Flow\Http\HttpRequestHandlerInterface;
use Neos\Flow\Mvc\ActionRequest;

// ...

/**
 * @var Bootstrap
 * @Flow\Inject
 */
protected $bootstrap;

// ...

$requestHandler = $this->bootstrap->getActiveRequestHandler();
if ($requestHandler instanceof HttpRequestHandlerInterface) {
    $actionRequest = ActionRequest::fromHttpRequest($requestHandler->
↪getHttpRequest());
    // ...
}
```

Arguments

The `ActionRequest` features a few methods for retrieving and setting arguments. These arguments are the result of merging any GET, POST and PUT arguments and even the information about uploaded files. Note that these arguments have already been processed by the validation and property mapping layerns and thus are suitable for being used in controller actions. If you, however, need to access the raw data, you can access these via the `getCookieParams()`, `getQueryParams()`, `getUploadedFiles()` and `getParsedBody()` methods of the `HttpRequest` respectively.

Arguments provided by POST or PUT requests are usually encoded in one or the other way. Flow detects the encoding through the `Content-Type` header and decodes the arguments and their values automatically into the parsed body.

getParsedBody()

You can access the request body easily by calling the `getParsedBody()` method. For performance reasons you may also retrieve the content as a stream instead of a parsed structure by calling `getBody()` before the `RequestBodyParsingComponent`. Please be aware though that, due to how input streams work in PHP, it is not possible to retrieve the content as a stream a second time, so the `RequestBodyParsingComponent` will not be able to parse the request body then.

Media Types

The best way to determine the media types mentioned in the `Accept` header of a request is to call the `\Neos\Flow\Http\Helper\MediaTypeHelper::determineAcceptedMediaTypes()` method. There is also a method implementing content negotiation in a convenient way: just pass a list of supported formats to `\Neos\Flow\Http\Helper\MediaTypeHelper::negotiateMediaType()` and in return you'll get the media type best fitting according to the preferences of the client:

```
$preferredType = \Neos\Flow\Http\Helper\MediaTypeHelper::negotiateMediaType(
        \Neos\Flow\Http\Helper\MediaTypeHelper::determineAcceptedMediaTypes($request),
        array('application/json', 'text/html') // These are the accepted media types
);
```

Request Methods

Flow supports all valid request methods, namely `CONNECT`, `DELETE`, `GET`, `HEAD`, `OPTIONS`, `PATCH`, `POST`, `PUT` and `TRACE`. Due to limited browser support and restrictive firewalls one sometimes need to tunnel request methods: By sending a `POST` request and specifying the `__method` argument, the request method can be overridden:

```
<form method="POST">
        <input type="hidden" name="__method" value="DELETE" />
</form>
```

Additionally Flow respects the `X-HTTP-Method` respectively `X-HTTP-Method-Override` header.

Trusted Proxies

If your server is behind a reverse proxy or a CDN, some of the request information like the the host name, the port, the protocol and the original client IP address are provided via additional request headers. Since those headers can also easily be sent by an adversary, possibly bypassing security measurements, you should make sure that those headers are only accepted from trusted proxies.

For this, you can configure a list of proxy IP address ranges in CIDR notation that are allowed to provide such headers, and which headers specifically are accepted for overriding those request information:

```
Neos:
  Flow:
    http:
      trustedProxies:
        proxies:
          - '216.246.40.0/24'
          - '216.246.100.0/24'

        headers:
```

(continues on next page)

(continued from previous page)

```
        clientIp: 'X-Forwarded-For'
        host: 'X-Forwarded-Host'
        port: 'X-Forwarded-Port'
        proto: 'X-Forwarded-Proto'
```

This would mean that only the `X-Forwarded-*` headers are accepted and only as long as those come from one of the IP ranges `216.246.40.0-255` or `216.246.100.0-255`. If you are using the standardized Forwarded Header, you can also simply set `trustedProxies.headers` to `'Forwarded'`, which is the same as setting all four properties to this value. By default, no proxies are trusted (unless the environment variable `FLOW_HTTP_TRUSTED_PROXIES` is set) and only the direct request informations will be used. If you specify trusted proxy addresses, by default only the `X-Forwarded-*` headers are accepted.

Note: On some container environments like ddev, the container acts as a proxy to provide port mapping and hence needs to be allowed in this setting. Otherwise the URLs generated will likely not work and end up with something along the lines of 'https://flow.ddev.local:80'. Therefore you probably need to set `Neos.Flow.http.trustedProxies.proxies` setting to '*' in your Development environment `Settings.yaml`.

You can also specify the list of IP addresses or address ranges in comma separated format, which is useful for using in the environment variable `FLOW_HTTP_TRUSTED_PROXIES`:

```
Neos:
  Flow:
    http:
      trustedProxies:
        proxies: '216.246.40.0/24,216.246.100.0/24'
```

Also, for backwards compatibility the following headers are trusted for providing the client IP address:

Client-Ip, X-Forwarded-For, X-Forwarded, X-Cluster-Client-Ip, Forwarded-For, Forwarded

Those headers will be checked from left to right and the first set header will be used for determining the client address.

Response

Being the counterpart to the request, the `Response` class represents the HTTP response. Its most important function is to contain the response body and the response status. Again, it is recommended to take a closer look at the actual class before you start using the API in earnest.

The `Response` class features a few specialities, we'd like to mention at this point:

Dates

The dates passed to one of the date-related methods must either be a RFC 2822 parsable date string or a PHP `DateTime` object. Please note that all methods returning a date will do so in form of a `DateTime` object.

According to RFC 2616 all dates must be given in Coordinated Universal Time, also known as `UTC`. UTC is also sometimes referred to as `GMT`, but in fact Greenwich Mean Time is not the correct time standard to use. Just to complicate things a bit more, according to the standards the HTTP headers will contain dates with the timezone declared as `GMT` – which in reality refers to `UTC`.

Flow will always return dates related to HTTP as UTC times. Keep that in mind if you pass dates from a different standard and then retrieve them again: the `DateTime` objects will mark the same point in time, but have a different time zone set.

Headers

Both classes, `Request` and `Response` inherit methods from the `Message` class. Among them are functions for retrieving and setting headers. If you need to deal with headers, please have a closer look at the `Headers` class which not only contains setters and getters but also some specialized cookie handling and cache header support.

In general, `Cache-Control` directives can be set through the regular `set()` method. However, a more convenient way to tweak single directives without overriding previously set values is the `setCacheControlDirective()` method. Here is an example – from the context of an Action Controller – for setting the `max-age` directive one hour:

```
$headers = $this->request->getHttpRequest()->getHeaders();
$headers->setCacheControlDirective('max-age', 3600);
```

Cookies

The HTTP foundation provides a very convenient way to deal with cookies. Instead of calling the PHP cookie functions (like `setcookie()`), we recommend using the respective methods available in the `ActionResponse` class.

Like requests and responses, a cookie also is represented by a PHP class. Instead of working on arrays with values, instances of the `Cookie` class are used. In order to set a cookie, just create a new `Cookie` object and add it to the HTTP response:

```
public function myAction() {
        $cookie = new Cookie('myCounter', 1);
        $this->response->setCookie($cookie);
}
```

As soon as the response is sent to the browser, the cookie is sent as part of it. With the next request, the user agent will send the cookie through the `Cookie` header. These headers are parsed automatically and can be retrieved from the `HttpRequest` object:

```
public function myAction() {
        $httpRequest = $this->request->getHttpRequest();
        $cookieParams = $httpRequest->getCookieParams();
        if (isset($cookieParams['myCounter']) {
                $this->view->assign('counter', (int)$cookieParams['myCounter']);
        }
}
```

The cookie value can be updated and re-assigned to the response:

```
public function myAction() {
        $httpRequest = $this->request->getHttpRequest();
        $counter = $httpRequest->getCookieParams()['myCounter'] ?? 0;
        $this->view->assign('counter', $counter);

        $cookie = new Cookie('myCounter', $counter + 1);
        $this->response->setCookie($cookie);
}
```

Finally, a cookie can be deleted by calling the `deleteCookie()` method:

```
public function myAction() {
        $this->response->deleteCookie('myCounter');
}
```

Uri

The `Http` sub package also provides a class representing a `Unified Resource Identifier`, better known as `URI`. The difference between a URI and a URL is not as complicated as you might think. "URI" is more generic, so URLs are URIs but not the other way around. A URI identifies a resource by its name or location. But it does not have to specify the representation of that resource – URLs do that. Consider the following examples:

A URI specifying a resource:

- http://flow.neos.io/images/logo

A URL specifying two different representations of that resource:

- http://flow.neos.io/images/logo.png
- http://flow.neos.io/images/logo.gif

Throughout the framework we use the term `URI` instead of `URL` because it is more generic and more often than not, the correct term to use.

All methods in Flow returning a URI will do so in form of a URI object. Most methods requiring a URI will also accept a string representation.

You are encouraged to use the `Uri` class for your own purposes – you'll get a nice API and validation for free!

Virtual Browser

The HTTP foundation comes with a virtual browser which allows for sending and receiving HTTP requests and responses. The browser's API basically follows the functions of a typical web browser. The requests and responses are used in form of `Http\Request` and `Http\Response` instances, similar to the requests and responses used by Flow's request handling mechanism.

Request Engines

The engine responsible for actually sending the request is pluggable. Currently there are two engines delivered with Flow:

- `InternalRequestEngine` simulates requests for use in functional tests
- `CurlEngine` uses the cURL extension to send real requests to other servers

Sending a request and processing the response is a matter of a few lines:

```
/**
 * A sample controller
 */
class MyController extends ActionController {

        /**
         * @Flow\Inject
         * @var \Neos\Flow\Http\Client\Browser
         */
        protected $browser;

        /**
         * @Flow\Inject
         * @var \Neos\Flow\Http\Client\CurlEngine
         */
```

(continues on next page)

(continued from previous page)

```
        protected $browserRequestEngine;

        /**
         * Some action
         */
        public function testAction() {
                $this->browser->setRequestEngine($this->browserRequestEngine);
                $response = $this->browser->request('https://www.flowframework.io');
                return ($response->hasHeader('X-Flow-Powered') ? 'yes' : 'no');
        }
}
```

As there is no default engine selected for the browser, you need to set one yourself. Of course you can use the advanced Dependency Injection techniques (through Objects.yaml) for injecting an engine into the browser you use.

Also note that the virtual browser is of scope Prototype in order to support multiple browsers with possibly different request engines.

Automatic Headers

The virtual browser allows for automatically sending specified headers along with every request. Simply pass the header to the browser as follows:

```
$browser->addAutomaticRequestHeader('Accept-Language', 'lv');
```

You can remove automatic headers likewise:

```
$browser->removeAutomaticRequestHeader('Accept-Language');
```

Functional Testing

The base test case for functional test cases already provides a browser which you can use for testing controllers and other application parts which are accessible via HTTP. This browser has the `InternalRequestEngine` set by default:

```
/**
 * Some functional tests
 */
class SomeTest extends \Neos\Flow\Tests\FunctionalTestCase {

        /**
         * @var boolean
         */
        protected $testableHttpEnabled = TRUE;

        /**
         * Send a request to a controller of my application.
         * Hint: The host name is not evaluated by Flow and thus doesn't matter
         *
         * @test
         */
        public function someTest() {
                $response = $this->browser->request('http://localhost/Acme.Demo/Foo/
↪bar.html');
```

(continues on next page)

(continued from previous page)

```
                $this->assertContains('it works', $response->getContent());
        }

}
```

2.3.8 Model View Controller

Flow promotes the use of the Model View Controller pattern which clearly separates the information, representation and mediation into separated building blocks. Although the design pattern and its naïve implementation are relatively simple, a capable MVC framework also takes care of more complex tasks such as input sanitizing, validation, form and upload handling and much more.

This chapter puts Flow's MVC framework into context with the HTTP request / response mechanism, explains how to develop controllers and describes various features of the framework.

HTTP

All action starts with an HTTP request sent from a client. The request contains information about the resource to retrieve or process, the action to take and various various parameters and headers. Flow converts the raw HTTP request into an HTTP Request object and, by invoking the *Routing* mechanism, determines which controller is responsible for processing the request and creating a matching response. A dispatcher then passes an internal to the controller and gets a response in return which can be sent to back to the client.

If you haven't done already, we recommend that you read the chapter about Flow's *HTTP Foundation*. It contains more detailed information about the application flow and the specific parts of the HTTP API.

Action Request

A typical application contains controllers providing one or more *actions*. While HTTP requests and responses are fine for communication between clients and servers, Flow uses a different kind of request internally to communicate with a controller, called `Action Request`. The default HTTP request handler asks the router to extract some information from the HTTP request and build an Action Request.

The Action Request contains the all the necessary details for calling the controller action which was requested by the client:

- the *package key* and optionally sub namespace of the package containing the controller supposed to handle the request
- the *controller name*
- the *action name*
- any *arguments* which are passed to the action
- the *format* of the expected response

With this information in place, the request handler can ask the `Dispatcher` to pass control to the specified controller.

Dispatcher

The Dispatcher has the function to invoke a controller specified in the given request and make sure that the request was processed correctly. The Dispatcher class provides one important method:

```
public function dispatch(RequestInterface $request, ResponseInterface $response) {
```

On calling this method, the Dispatcher resolves the controller class name of the controller mentioned in the request object and calls its `processRequest()` method. A fresh `Response` object is also passed to the controller which is expected to deliver its response data by calling the respective setter methods on that object.

Each request carries a `dispatched` flag which is set or unset by the controller. The Action Controller for example sets this flag by default and only unsets it if an action initiated a forward to another action or controller. If the flag is not set, the Dispatcher assumes that the request object has been updated with a new controller, action or arguments and that it should try again to dispatch the request. If dispatching the request did not succeed after several trials, the Dispatcher will throw an exception.

Sub Requests

An `Http\Request` object always reflects the original HTTP request sent by the client. It is not possible to create an *HTTP* sub request because requests which are passed along within the application must be instances of `Mvc\ActionRequest`. Creating an Action Request as a sub request of the original HTTP Request is simple, although you rarely need to do that:

```
$actionRequest = new ActionRequest($httpRequest);
```

An Action Request always holds a reference to a *parent request*. In most cases the hierarchy is shallow and the Action Request is just a direct sub request of the HTTP Request. In the context of a controller, it is easy to get a hold of the parent request:

```
public function fooAction() {
        $parentRequest = $this->request->getParentRequest();
        $httpRequest = $this->request->getHttpRequest();
        // in case of a shallow hierarchy, $parentRequest == $httpRequest
}
```

In a more complex scenario, an Action Request can be a sub request of another Action Request. This is the case in most implementations of plugins, widgets or other inline elements of a rendered page because each of them is a part of the whole and can be arbitrarily nested. Each element (plugin, widget ...) needs its own Action Request instance in order to keep track of invocation details like arguments and other context information.

A sub request can be created manually by passing the parent request to the constructor of the new Action Request:

```
$subRequest = new ActionRequest($parentRequest);
```

The top level Action Request (just below the HTTP Request) is referred to as the *Main Request*:

```
public function fooAction() {
        $parentRequest = $this->request->getParentRequest();
        $httpRequest = $this->request->getHttpRequest();
        $mainRequest = $this->request->getMainRequest();

        if ($this->request === $mainRequest) {
                $message = 'This is the main request';
        }
```

(continues on next page)

(continued from previous page)

```
                // same like above:
        if ($this->request->isMainRequest()) {
                $message = 'This is the main request';
        }
}
```

Manual creation of sub requests is rarely necessary. In most cases the framework will take care of creating and managing sub requests if plugins or widgets are in the game.

Controllers

A controller is responsible for preparing a model and collecting the necessary data which should be returned as a response. It also controls the application flow and decided if certain operations should be executed and how the application should proceed, for example after the user has submitted a form.

A controller should only sparingly contain logic which goes beyond these tasks. Operations which belong to the domain of the application should be rather be implemented by *domain services*. This allows for a clear separation of application flow and business logic and enables other parts of the application (for example web services) to execute these operations through a well-defined API.

A controller suitable for being used in Flow needs to implement the `Mvc\Controller\ControllerInterface`. At the bare minimum it must provide a `processRequest()` method which accepts a request and response.

If needed, custom controllers can be implemented in a convenient way by extending the `Mvc\Controller\AbstractController` class. The most common case though is to use the *Action Controller* provided by the framework.

Action Controller

Most web applications will interact with the client through execution of specific *actions* provided by an Action Controller. Flow provides a base class which contains all the logic to map and validate arguments found in the raw request to method arguments of an action. It also provides various convenience methods which are typically needed in Action Controller implementations.

A Simple Action

The most simple way to implement an action is to extend the ActionController class, declare an action method and return a plain string as the response:

```
namespace Acme\Demo\Controller;
use Neos\Flow\Mvc\Controller\ActionController;

class HelloWorldController extends ActionController {

        /**
         * The default action of this controller.
         *
         * @return string
         */
        public function indexAction() {
                return 'Hello world.';
        }
```

(continues on next page)

(continued from previous page)

```
}
```

Note that the controller must reside in the `Controller` sub namespace of your package in order to be detected by the default routing configuration. In the example above, `Acme\Demo` corresponds with the package key `Acme.Demo`.

By convention, `indexAction` is the action being called if no specific action was requested. An action method name must be camelCased and always end with the suffix "Action". In the Action Request and other parts of the routing system, it is referred to simply by its *action name*, in this case `index`.

If an action returns a string or an object which can be cast to a string, a PHP resource stream like an opened file or a PSR7 stream, it will be set as the content of the response automatically:

```
/**
 * Stream a file content to the browser
 *
 * @return resource
 */
public function exportAction() {
        $this->response->setContentType('text/csv');
        return fopen('/path/fo/file', 'r');
}
```

Note: This also works for large files, in which case the file content will be streamed to the browser.

Defining Arguments

The unified arguments sent through the HTTP request (that includes query parameters from the URI, possible POST arguments and uploaded files) are pre-processed and mapped to method arguments of an action. That means: all arguments a action needs in order to work should be declared as *method parameters* of the action method and not be retrieved from one of the superglobals ($_GET, $_POST, ...) or the HTTP request.

Declaring arguments in an action controller is very simple:

```
/**
 * Says hello to someone.
 *
 * @param string $name Name of the someone
 * @param boolean $formal If the message should be formal (or casual)
 * @return string
 */
public function sayHelloAction($name, $formal = TRUE) {
        $message = ($formal ? 'Greetings, Mr. ' : 'Hello, ') . $name;
        return $message
}
```

The first argument `$name` is mandatory. The `@param` annotation gives Flow a hint of the expected type, in this case a string.

The second argument `$boolean` is optional because a default value has been defined. The `@param` annotation declares this argument to be a boolean, so you can expect that `$formal` will be, in any case, either `TRUE` or `FALSE`.

A simple way to pass an argument to the action is through the query parameters in a URL:

```
http://localhost/acme.demo/helloworld/sayhello.html?name=Robert&formal=0
```

Note: Please note that the documentation block of the action method is mandatory – the annotations (tags) you see in the example are important for Flow to recognize the correct type of each argument.

Additionally to passing the arguments to the action method, all registered arguments are also available through `$this->arguments`.

Argument Mapping

Internally the Action Controller uses the Property Mapper for mapping the raw arguments of the HTTP request to an `Mvc\Controller\Arguments` object. The Property Mapper can convert and validate properties while mapping them, which allows for example to transparently map values of a submitted form to a new or existing model instance. It also makes sure that validation rules are considered and that only certain parts of a nested object structure can be modified through user input.

In order to understand the mapping process, we recommend that you take a look at the respective chapter about *Property Mapping*.

Here are some more examples illustrating the mapping process of submitted arguments to the method arguments of an action:

Besides simple types, also special object types, like `DateTime` are supported:

```
# http://localhost/acme.demo/foo/bar.html?date=2012-08-10T14:51:01+02:00

/**
 * @param \DateTime $date Some date
 * @return string
 */
public function barAction(\DateTime $date) {
        # ...
}
```

Properties of domain models (or any other objects) can be set through an array-like syntax. The property mapper creates a new object by default:

```
# http://localhost/acme.demo/foo/create.html?customer[name]=Robert

/**
 * @param Acme\Demo\Domain\Model\Customer $customer A new customer
 * @return string
 */
public function createAction(\Acme\Demo\Domain\Model\Customer $customer) {
        return 'Hello, new customer: ' . $customer->getName();
}
```

If an identity was specified, the Property Mapper will try to retrieve an object of that type:

```
# http://localhost/acme.demo/foo/create.html?customer[number]=42&customer[name]=Robert

/**
 * @param Acme\Demo\Domain\Model\Customer $customer An existing customer
 * @param string $name The name to set
```

(continues on next page)

(continued from previous page)

```
     * @return string
     */
    public function updateAction(\Acme\Demo\Domain\Model\Customer $customer, $name) {
            $customer->setName($name);
            $this->customerRepository->update($customer);
    }
```

Note: `number` must be declared as (part of) the identity of a `Customer` object through an `@Identity` annotation. You'll find more information about identities and also about the creation and update of objects in the *Persistence* chapter.

Instead of passing the arguments through the query string, like in the previous examples, they can also be submitted as POST or PUT arguments in the body of a request or even be a mixture of both, query parameters and parameters contained in the HTTP body. Argument values are merged in the following order, while the later sources replace earlier ones

- query string (derived from $_GET)
- body (typically from POST or PUT requests)
- file uploads (derived from $_FILES)

Hint: Sometimes you might need to map a whole request body into a single action argument. In that case you can use an annotation `@Flow\MapRequestBody("$argumentName")` on your action. Please refer to the *Property Mapping* chapter for more details.

Internal Arguments

In some situations Flow needs to set special arguments in order to simplify handling of objects, widgets or other complex operations. In order to avoid name clashes with arguments declared by a package author, a special prefix consisting of two underscores __ is used. Two examples of internal arguments are the automatically generated *HMAC* and *CSRF* hashes[1] which are sent along with the form data:

```
<form enctype="multipart/form-data" name="newPost" method="post"
                action="posts/create">
        <input type="hidden" name="__trustedProperties" value="a:3:{s:4:&amp;quot;blog&
↪quot;;...
        <input type="hidden" name="__csrfToken" value="__
↪csrfToken=cca240aa13af5bdacea3...
        <label for="author">Author</label><br />
        <input id="author" type="text" name="newPost[author]" value="First Last" />
↪<br />
        ...
```

Although internal arguments can be retrieved through a method provided by the `ActionRequest` object, they are, as the name suggests, only for internal use. You should not use or rely on these arguments in your own applications.

[1] The HMAC and CSRF hashes improve security for form submissions and actions on restricted resources. Please refer to the *Security* chapter for more details.

Plugin Arguments

Besides internal arguments, Flow stores arguments being used by recursive controller invocations, like plugins, in a separate namespace, the so called `pluginArguments`.

They are prefixed with two dashes `--` and normally, you do not interact with them.

initialize*()

The Action Controller's `processRequest()` method initializes important parts of the controller, maps and validates arguments and finally calls the requested action method. In order to execute code before the action method is called, it is possible to implement one or more initialization methods. The following methods are currently supported:

- `initializeAction()`
- `initialize[ActionName]()`
- `initializeView()`

The first method executed after the base initialization is `initializeAction()`. The Action Controller only provides an empty method which can be overriden by a concrete Action Controller. The information about action method arguments and the corresponding validators has already been collected at this point, but any arguments sent through the request have not yet been mapped or validated. Therefore, `initializeAction()` can still modify the list of possible arguments or add / remove certain validators by altering `$this->arguments`.

Right after the generic `initializeAction()` method has been called, the Action Controller checks if a more specific initialization method was implemented. For example, if the action name is "create" and thus the action method name is `createAction()`, the controller would try to call a method `initializeCreateAction()`. This allows for execution of code which is targeted directly to a specific action.

Finally, after arguments have been mapped and the controller is almost ready to call the action method, it tries to resolve a suitable *view* and, if it was successful, runs the `initializeView()` method. In many applications, the view implementation will be a Fluid Template View. The `initializeView()` method can be used to assign template variables which are needed in any of the existing actions or conduct other template-specific configuration steps.

Media Type / Format

Any implementation based on `AbstractController` can support one or more formats for its response. Depending on the preferences of the client sending the request and the route which matched the request the controller needs render the response in a format the client understands.

The supported and requested formats are specified as an IANA Media Type and is, by default, `text/html`. In order to support a different or more than one media type, the controller needs override the default simply by declaring a class property like in the following example:

```
class FooController extends ActionController {

        /**
         * A list of IANA media types which are supported by this controller
         *
         * @var array
         */
        protected $supportedMediaTypes = array('application/json', 'text/html');

        # ...
}
```

The media types listed in `$supportedMediaTypes` don't need to be in any particular order.

The Abstract Controller determines the preferred format through Content Negotiation. More specifically, Flow will check if any specific format was defined in the route which matched the request (see chapter *Routing*). If no particular format was defined, the `Accept` header of the HTTP Request is consulted for a weighted list of preferred media types. This list is then matched with the list of supported media types and hopefully results in one media type which is set as the `format` in the Action Request.

Hint: With "format" we are referring to the typical file extension which corresponds to a specific media type. For example, the format for `text/html` is "html" and the format corresponding to the media type `application/json` would be "json". For a complete list of supported media types and their corresponding formats please refer to the class `Neos\Utility\MediaTypes`.

The controller implementation must take care of the actual media type support by supplying a corresponding view or template.

Fluid Template View

An Action Controller can directly return the rendered content by means of a string returned by the action method. However, this approach is not very flexible and ignores the separation of concerns as laid out by the Model View Controller pattern. Instead of rendering an output itself, a controller delegates this task to a view.

Flow uses the Fluid template engine as the default view for action controllers. By following a naming convention for directories and template files, developers of a concrete controller don't need to configure the view or paths to the respective templates – they are resolved automatically by converting the combination of package key, controller name and action name into a Fluid template path.

Given that the package key is `Acme.Demo`, the controller name is `HelloWorld`, the action name is `sayHello` and the format is `html`, the following path and filename would be used for the corresponding Fluid template:

```
./Packages/.../Acme.Demo/Resources/Private/Templates/HelloWorld/SayHello.html
```

If a template file matching the current request was found, the Action Controller initializes a Fluid Template View with the correct path name. This pre-initialized view is available via `$this->view` in any Action Controller and can be used for assigning template variables:

```
$this->view->assign('products', $this->productRepository->findAll());
```

If an action does not return a result (that is, the result is `NULL`), an Action Controller automatically calls the `render()` method of the current view. That means, apart from assigning variables to the template (if any), there is rarely a need to deal further with a Fluid Template View.

Json View

When used as a web service, controllers may want to return data in a format which can be easily used by other applications. Especially in a web context JSON has become an often used format which is very light-weight and easy to parse. Although it is theoretically possible to render a JSON response through a Fluid Template View, a specialized view does a much better job in a more convenient way.

The JSON View provided by Flow can be used by declaring it as the default view in the concrete Action Controller implementation:

```
class FooController extends ActionController {

        /**
         * @var string
         */
        protected $defaultViewObjectName = \Neos\Flow\Mvc\View\JsonView::class;

        # ...
}
```

Alternatively, if more than only the JSON format should be supported, the format to view mapping feature can be used:

```
class FooController extends ActionController {

        /**
         * @var string
         */
        protected $viewFormatToObjectNameMap = array(
                'html' => \Neos\FluidAdaptor\View\TemplateView::class,
                'json' => \Neos\Flow\Mvc\View\JsonView::class
        );

        /**
         * A list of IANA media types which are supported by this controller
         *
         * @var array
         */
        protected $supportedMediaTypes = array('application/json', 'text/html');

        # ...
}
```

In either case, the JSON View is now invoked if a request is sent which prefers the media type `application/json`. In order to return something useful, the data which should be rendered as JSON must be set through the `assign()` method. By default JSON View uses the variable named "value":

```
/**
 * @param \Acme\Demo\Model\Product $product
 * @return void
 */
public function showAction(Product $product) {
        $this->view->assign('value', $product);
}
```

To change the name of the rendered variables, use the `setVariablesToRender()` method on the view.

If the controller is configured to use the JSON View, this action may return JSON code like the following:

```
{"name":"Arabica","weight":1000,"price":23.95}
```

Furthermore, the JSON view can be configured to determine which variables of the object should be included in the output. For that, a configuration array needs to be provided with `setConfiguration()`:

```
/**
 * @param \Acme\Demo\Model\Product $product
 * @return void
```

(continues on next page)

(continued from previous page)

```
 */
public function showAction(Product $product) {
        $this->view->assign('value', $product);
        $this->view->setConfiguration(/* configuration follows here */);
}
```

The configuration is an array which is structured like in the following example:

```
array(
        'value' => array(

                        // only render the "name" property of value
                '_only' => array('name')
        ),
        'anothervalue' => array(

                        // render every property except the "password"
                        // property of anothervalue
                '_exclude' => array('password')

                        // we also want to include the sub-object
                        // "address" as nested JSON object
                '_descend' => array(
                        'address' => array(
                                // here, you can again configure
                                // _only, _exclude and _descend if needed
                        )
                )
        ),
        'arrayvalue' => array(

                        // descend into all array elements
                '_descendAll' => array(
                        // here, you can again configure _only,
                        // _exclude and _descend for each element
                )
        ),
        'valueWithObjectIdentifier' => array(

                        // by default, the object identifier is not
                        // included in the output, but you can enable it
                '_exposeObjectIdentifier' => TRUE,

                        // the object identifier should not be rendered
                        // as "__identity", but as "guid"
                '_exposedObjectIdentifierKey' => 'guid'
        )
)
```

To sum it up, the JSON view has the following configuration options to control the output structure:

- `_only` (array): Only include the specified property names in the output
- `_exclude` (array): Include all except the specified property names in the output
- `_descend` (associative array): Descend into the specified sub-objects
- `_descendAll` (array): Descend into all array elements and generate a numeric array

- `_exposeObjectIdentifier` (boolean): if TRUE, the object identifier is displayed inside `__identifier`
- `_exposeObjectIdentifierKey` (string): the JSON field name inside which the object identifier should be displayed

Custom View

Similar to the Fluid Template View and the JSON View, packages can provide their own custom views. The only requirement for such a view is the implementation of all methods defined in the `Neos\Flow\Mvc\View\ViewInterface`.

An Action Controller can be configured to use a custom view through the `$defaultViewObjectName` and `$viewFormatToObjectNameMap` properties, as explained in the section about JSON View.

Configuring Views through Views.yaml

If you want to change Templates, Partials, Layouts or the whole ViewClass for a foreign package without modifying it directly, and thus breaking updatability, you can create a `Views.yaml` in your configuration folder and override all options the view supports.

The general syntax of a view configuration looks like this:

```
-
  requestFilter: 'isPackage("Foreign.Package") && isController("Standard")'
  viewObjectName: 'Neos\Fusion\View\FusionView'
  options:
    fusionPathPatterns:
      - 'resource://Neos.Fusion/Private/Fusion'
      - 'resource://My.Package/Private/Fusion'
    fusionPath: 'yourPrototype'
```

The requestFilter is based on Neos.Eel allowing you to match arbitrary requests so that you can override View configuration for various scenarios. You can combine any of these matchers to filter as specific as you need:

- isPackage("Package.Key")
- isSubPackage("SubPackage")
- isController("Standard")
- isAction("index")
- isFormat("html")

There are additional helpers to get the parentRequest or mainRequest of the current request, which you can use to limit some configuration to only take effect inside a specific subRequest. All Eel matchers above can be used with the parentRequest or mainRequest as well:

- parentRequest.isPackage("Neos.Neos")
- parentRequest.isController("Standard")
- mainRequest.isController("Standard")
- ...

You can combine any of these matchers with boolean operators:

(isPackage("My.Foo") || isPackage('My.Bar')) && isFormat("html")

The order of the configurations is in most cases unimportant. Each matcher has a specific weight similar to CSS specifity (ID, class, inline, important) to determine which configuration outweighs the other. For each match resulting matcher the weight will be increased by a certain value.

Method	Weight
isPackage("Package.Key")	1
isSubPackage("SubPackage")	10
isController("Standard")	100
isAction("index")	1000
isFormat("html")	10000
mainRequest()	100000
parentRequest()	1000000

If the package is "My.Foo" and the Format is "html" the result will be 10001

Note: Previously the configuration of all matching `Views.yaml` filters was merged. From version 4.0 on only the matching filter with the highest weight is respected in order to reduce ambiguity.

The fusionPathPatterns has to contain the Root-Fusion and the path to Fusion-Folder which contains your Prototype. Your Prototype gets searched recursively by fusionPath.

Controller Context

The Controller Context is an object which encapsulates all the controller-related objects and makes them accessible to the view. Thus, the `$this->request` property of the controller is available inside the view as `$this->controllerContext->getRequest()`.

Validation

Arguments which were sent along with the HTTP request are usually sanitized and valdidated before they are passed to an action method of a controller. Behind the scenes, the *Property Mapper* is used for mapping and validating the raw input. During this process, the validators are invoked:

- *base validation* as defined in the model to be validated (if any)
- *argument validation* as defined in the controller or action

The chapter about *Validation* outlines the general validation mechanism and how declare and configure *base validation*. While the rules declared in a model describe the minimum requirements for a valid entity, the rules declared in a controller define additional preconditions before arguments may be passed to an action method.

Per-action validation rules are declared through the `Validate` annotation. As an example, an email address maybe optional in a Customer model, but it may be required when a customer entity is passed to a `signUpAction()` method:

```
/**
 * @param \Acme\Demo\Domain\Model\Customer $customer
 * @Flow\Validate(argumentName="emailAddress", type="EmailAddress")
 */
public function signUpAction(Customer $customer) {
        # ...
}
```

While `Validate` defines additional rules, the `IgnoreValidation` annotation does the opposite: any base validation rules declared for the specified argument will be ignored:

```
/**
 * @param \Acme\Demo\Domain\Model\Customer $customer
 * @Flow\IgnoreValidation("$customer")
 */
public function signUpAction(Customer $customer) {
        # ...
}
```

By default the validation for an argument annotated with `IgnoreValidation` will not be executed. If the result is needed for further processing in the action method, the `evaluate` flag can be enabled:

```
/**
 * @param \Acme\Demo\Domain\Model\Customer $customer
 * @Flow\IgnoreValidation("$customer", evaluate=true)
 */
public function signUpAction(Customer $customer) {
        if ($this->arguments['customer']->getValidationResults()->hasErrors()) {
                # ...
        }
}
```

The next section explains how to get a hold of the validation results and react on warnings or errors which occurred during the mapping and validation step.

Error Handling

The argument mapping step based on the validation rules mentioned earlier makes sure that an action method is only called if its arguments are valid. In the reverse it means that the action specified by the request will not be called if a mapping or validation error occurred. In order to deal with these errors and provide a meaningful error message to the user, a special action is called instead of the originally intended action.

The default implementation of the `errorAction()` method will redirect the browser to the URI it came from, for example to redisplay the originally submitted form.

Any errors or warnings which occurred during the argument mapping process are stored in a special object, the *mapping results*. These mapping results can be conveniently access through a Fluid view helper in order to display warnings and errors along the submitted form or on top of it:

```
<f:validation.results>
        <f:if condition="{validationResults.flattenedErrors}">
                <ul class="errors">
                        <f:for each="{validationResults.flattenedErrors}" as="errors"␣
↪key="propertyPath">
                                <li>{propertyPath}
                                        <ul>
                                                <f:for each="{errors}" as="error">
                                                        <li>{error.code}: {error}</li>
                                                </f:for>
                                        </ul>
                                </li>
                        </f:for>
                </ul>
        </f:if>
</f:validation.results>
```

Besides using the view helper to display the validation results, you can also completely replace the `errorAction()` method with your own custom method.

Upload Handling

The handling of file uploads is pretty straight forward. Files are handled internally as `PersistentResource` objects and thus, storing an uploaded file is just a matter of declaring a property of type `PersistentResource` in the respective model.

There is a full example explaining file uploads in the *chapter about resource management*.

REST Controller

tbd.

Generating Links

Links to other controller and their actions should not be rendered manually because hardcoded or manually rendered links circumvent many of Flow's features.

For generating links to other controllers, the `UriBuilder` which is available as `$this->uriBuilder` can be used. However, in most cases, the user does not directly interact with this one, but rather uses `forward()`, `redirect()` in the Controller and `<f:link.action />` / `<f:uri.action />` inside Fluid templates.

forward() and redirect()

Often, controllers need to defer execution to other controllers or actions. For that to happen, Flow supports both, internal and external redirects:

- in an internal redirect which is triggered by `forward()`, the URI does not change.
- in an external redirect, the browser receives a HTTP `Location` header, redirecting him to the new controller. Thus, the URI changes.

As a consequence, `forward()` can also call controllers or actions which are not exposed through the routing mechanism, while `redirect()` only works with publicly callable controllers.

This example demonstrates the usage of `redirect()`:

```
public function createAction(Product $product) {
                // TODO: store the product somewhere

        $this->redirect('show', NULL, NULL, array('product' => $product));

                // This line is never executed, as redirect() and
                // forward() immediately stop execution of this method.
}
```

It is good practice to have different actions for *modifying* and *showing* data. Often, redirects are used to link between them. As an example, an `updateAction()` which updates an object should then `redirect()` to the `show` action of the controller, then displays the updated object.

`forward()` supports the following arguments:

- `$actionName` (required): Name of the target action
- `$controllerName`: Name of the target controller. If not specified, the current controller is used.

- `$packageKey`: Name of the package, optionally with sub-package. If not specified, the current package key / subpackage key is specified. The package and sub-package need to be delimited by `\`, so `Foo.Bar\Test` will set the package to `Foo.Bar` and the subpackage to `Test`.
- `$arguments`: array of request arguments. Objects are automatically converted to their identity.

`redirect()` supports all of the above arguments, additionally with the following ones:

- `$delay`: Delay in seconds before redirecting
- `$statusCode`: the status code to be used for redirecting. By default, 303 is used.
- `$format`: The target format for the redirect. If not set, the current format is used.

Flash Messages

In many applications users need to be notified about the application flow, telling him for example that an object has been successfully saved or deleted. Such messages, which should be displayed to the user only once, are called *Flash Messages*.

A Flash Message can be added inside the controller by using the `addFlashMessage` method, which expects the following arguments:

- `$messageBody` (required): The message which should be shown
- `$messageTitle`: The title of the message
- `$severity`: The severity of the message; by default "OK" is used. Needs to be one of NeosErrorMessagesMessage::SEVERITY_* constants (OK, NOTICE, WARNING, ERROR)
- `$messageArguments` (array): If the message contains any placeholders, these can be filled here. See the PHP function `printf` for details on the placeholder format.
- `$messageCode` (integer): unique code of this message, can be used f.e. for localization. By convention, if you set this, it should be the UNIX timestamp at time of writing the source code to be roughly unique.

Creating a Flash Messages is a matter of a single line of code:

```
$this->addFlashMessage('Everything is all right.');
$this->addFlashMessage('Sorry, I messed it all up!', 'My Fault', \Neos\Error\Messages\
↪Message::SEVERITY_ERROR);
```

The flash messages can be rendered inside the template using the `<f:flashMessages />` ViewHelper. Please consult the ViewHelper for a full reference.

Since Flash Messages need to possibly survive over requests until they get displayed, they need to be persisted somehow. Flash Messages can be stored in different ways, the Framework default is to store them in the session. The storage can be configured in Settings.yaml via the following options:

```
Neos:
  Flow:
    mvc:
      flashMessages:
        containers:
          'customFlashMessages':
            storage: 'Neos\Flow\Mvc\FlashMessage\Storage\FlashMessageCookieStorage'
            storageOptions:
              cookieName: 'Neos_Flow_FlashMessages_My_Custom'
            requestPatterns:
              'SomeControllers':
                pattern: 'ControllerObjectName'
```

(continues on next page)

(continued from previous page)

```
                patternOptions:
                  'controllerObjectNamePattern': 'Some\Package\Controller\.*'
```

With this you can specify to store the Flash Messages in an own cookie, and even separate them by request patterns. New storages can be created by implementing the `FlashMessageStorageInterface` and specifying the storage class in the settings.

2.3.9 Templating

Templating is done in *Fluid*, which is a next-generation templating engine. It has several goals in mind:

- Simplicity
- Flexibility
- Extensibility
- Ease of use

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

What Does it Do?

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

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

Below, you'll find a snippet of a real-world template displaying a list of blog postings. Use it to check whether you find the template language intuitive:

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

- The *Namespace Import* makes the `\Neos\FluidAdaptor\ViewHelper` namespace available under the shorthand f.
- The `<f:for>` essentially corresponds to `foreach ($postings as $posting)` in PHP.

- With the dot-notation (`{posting.title}` or `{posting.author.name}`), you can traverse objects. In the latter example, the system calls `$posting->getAuthor()->getName()`.
- The `<f:link.action />` tag is a so-called ViewHelper. It calls arbitrary PHP code, and in this case renders a link to the "details"-Action.

There is a lot more to show, including:

- Layouts
- Custom View Helpers
- Boolean expression syntax

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

Basic Concepts

This section describes all basic concepts available. This includes:

- Namespaces
- Variables / Object Accessors
- View Helpers
- Arrays

Namespaces

Fluid can be extended easily, thus it needs a way to tell where a certain tag is defined. This is done using namespaces, closely following the well-known XML behavior.

Namespaces can be defined in a template in two ways:

{namespace f=NeosFluidAdaptorViewHelpers} This is a non-standard way only understood by Fluid. It links the `f` prefix to the PHP namespace `\Neos\FluidAdaptor\ViewHelpers`.

<html xmlns:foo="http://some/unique/namespace"> The standard for declaring a namespace in XML. This will link the `foo` prefix to the URI `http://some/unique/namespace` and Fluid can look up the corresponding PHP namespace in your settings (so this is a two-piece configuration). This makes it possible for your XML editor to validate the template files and even use an XSD schema for auto completion.

A namespace linking `f` to `\Neos\FluidAdaptor\ViewHelpers` is imported by default. All other namespaces need to be imported explicitly.

If using the XML namespace syntax the default pattern `http://typo3.org/ns/<php namespace>` is resolved automatically by the Fluid parser. If you use a custom XML namespace URI you need to configure the URI to PHP namespace mapping. The YAML syntax for that is:

```
Neos:
  Fluid:
    namespaces:
      'http://some/unique/namespace': 'My\Php\Namespace'
```

Variables and Object Accessors

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

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

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

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

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

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

Note: The semantics between the controller and the view should be the following: The controller instructs the view to "render the blog object given to it", and not to "render the Blog title, and the blog posting 1, . . . ".

Passing objects to the view instead of simple values is highly encouraged!

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

```
// This should go into the controller:
$this->view->assign('blog', $blog);
```

```
<!-- This should go into the template: -->
<h1>This blog is called {blog.title}, written by {blog.author}</h1>
```

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

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

View Helpers

All output logic is placed in View Helpers.

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

This concept is best understood with an example:

```
{namespace f=Neos\FluidAdaptor\ViewHelpers}
<f:link.action controller="Administration">Administration</f:link.action>
```

The example consists of two parts:

- *Namespace Declaration* as explained earlier.
- *Calling the View Helper* with the `<f:link.action...> ... </f:link.action>` tag renders a link.

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

The `<f:link.action />` tag is implemented in the class `\Neos\FluidAdaptor\ViewHelpers\Link\ActionViewHelper`.

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

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

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

Note: Some benefits of the class-based approach approach are:

- You cannot override already existing view helpers by accident.
- It is very easy to write custom view helpers, which live next to the standard view helpers
- All user documentation for a view helper can be automatically generated from the annotations and code documentation.

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

This is often used when adding arguments to links:

```
<f:link.action controller="Blog" action="show" arguments="{singleBlog: blogObject}">
  ... read more
</f:link.action>
```

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

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

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

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

Boolean Expressions

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

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

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

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

All boolean expressions have the form `X <comparator> Y`, where:

- *<comparator>* is one of the following: `==, >, >=, <, <=, % (modulo)`
- *X* and *Y* are one of the following:
  - a number (integer or float)
  - a string (in single or double quotes)
  - a JSON array
  - a ViewHelper
  - an Object Accessor (this is probably the most used example)
  - inline notation for ViewHelpers

Inline Notation for ViewHelpers

In many cases, the tag-based syntax of ViewHelpers is really intuitive, especially when building loops, or forms. However, in other cases, using the tag-based syntax feels a bit awkward – this can be demonstrated best with the `<f:uri.resource>`- ViewHelper, which is used to reference static files inside the *Public/* folder of a package. That's why it is often used inside `<style>` or `<script>`-tags, leading to the following code:

```
<link rel="stylesheet" href="<f:uri.resource path='myCssFile.css' />" />
```

You will notice that this is really difficult to read, as two tags are nested into each other. That's where the inline notation comes into play: It allows the usage of `{f:uri.resource()}` instead of `<f:uri.resource />`. The above example can be written like the following:

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

This is readable much better, and explains the intent of the ViewHelper in a much better way: It is used like a helper function.

The syntax is still more flexible: In real-world templates, you will often find code like the following, formatting a `DateTime` object (stored in `{post.date}` in the example below):

```
<f:format.date format="d-m-Y">{post.date}</f:format.date>
```

This can also be re-written using the inline notation:

```
{post.date -> f:format.date(format:'d-m-Y')}
```

This is also a lot better readable than the above syntax.

Tip: This can also be chained indefinitely often, so one can write:

```
{post.date -> foo:myHelper() -> bar:bla()}
```

Sometimes you'll still need to further nest ViewHelpers, that is when the design of the ViewHelper does not allow that chaining or provides further arguments. Have in mind that each argument itself is evaluated as Fluid code, so the following constructs are also possible:

```
{foo: bar, baz: '{planet.manufacturer -> f:someother.helper(test: \'stuff\')}'}
{some: '{f:format.stuff(arg: \'foo'\)}'}
```

To wrap it up: Internally, both syntax variants are handled equally, and every ViewHelper can be called in both ways. However, if the ViewHelper "feels" like a tag, use the tag-based notation, if it "feels" like a helper function, use the Inline Notation.

Arrays

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

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

The array syntax used here is very similar to the JSON object syntax. Thus, the left side of the associative array is used as key without any parsing, and the right side can be either:

- a number:

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

- a string; Needs to be in either single- or double quotes. In a double-quoted string, you need to escape the `"` with a `\` in front (and vice versa for single quoted strings). A string is again handled as Fluid Syntax, this is what you see in example `c`:

  ```
  {a : 'Hallo',
   b : "Second string with escaped \" (double quotes) but not escaped ' (single␣
  ↪quotes)"
   c : "{firstName} {lastName}"
  }
  ```

- a boolean, best represented with their integer equivalents:

```
{a : 'foo',
 notifySomebody: 1
 useLogging: 0
}
```

- a nested array:

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

- a variable reference (=an object accessor):

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

Note: All these array examples will result into an associative array. If you have to supply a non-associative, i.e. numerically-indexed array, you'll write `{0: 'foo', 1: 'bar', 2: 'baz'}`.

Passing Data to the View

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

Passing data to the view from outside a controller

You can also pass data to the view from outside a controller. This can be useful for general data, that you want to be available without having to assign it in each action.

Once the view is resolved inside the `ActionController`, the signal `viewResolved` is being emitted and you can add data.

This is possible with the Signal/Slot dispatcher from your `Package.php` file:

```
<?php
namespace Vendor\Namespace;

use Neos\Flow\Core\Bootstrap;
use Neos\Flow\Mvc\Controller\ActionController;
use Neos\Flow\Mvc\View\ViewInterface;
use Neos\Flow\Package\Package as BasePackage;


/**
 * The Flow Package
 */
class Package extends BasePackage
{
```

(continues on next page)

(continued from previous page)

```
    /**
     * Invokes custom PHP code directly after the package manager has been␣
↪initialized.
     *
     * @param Bootstrap $bootstrap The current bootstrap
     * @return void
     */
    public function boot(Bootstrap $bootstrap)
    {

        $dispatcher = $bootstrap->getSignalSlotDispatcher();

        $dispatcher->connect(ActionController::class, 'viewResolved', static function␣
↪(ViewInterface $view) {
            $view->assign('settingPassedFromSignal', 'sun is shining');
        });

    }
}
```

Layouts

In almost all web applications, there are many similarities between each page. Usually, there are common templates or menu structures which will not change for many pages.

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

Writing a Layout

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

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

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

Using a Layout

Using a layout involves two steps:

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

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

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

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

Writing Your Own ViewHelper

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

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

```
public function render()
```

Rendering the View Helper

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

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

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

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

Note: If you add variables to the `TemplateVariableContainer`, make sure to remove every variable which you added again. This is a security measure against side-effects.

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

Implementing a `for` ViewHelper

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

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

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

So, in words, what should the loop do?

It needs two arguments:

- `each`: Will be set to some object or array which can be iterated over.
- `as`: The name of a variable which will contain the current element being iterated over

It then should do the following (in pseudo code):

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

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

```
class ForViewHelper extends \Neos\FluidAdaptor\Core\ViewHelper\AbstractViewHelper {

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

}
```

- The PHPDoc is part of the code! Fluid extracts the argument data types from the PHPDoc.
- You can simply register arguments to the view helper by adding them as method arguments of the `render()` method.
- Using `$this->renderChildren()`, everything between the opening and closing tag of the view helper is rendered and returned as string.

Declaring Arguments

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

You can also register arguments inside a method called `initializeArguments()`. Call `$this->registerArgument($name, $dataType, $description, $isRequired, $defaultValue=NULL)` inside.

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

AbstractTagBasedViewHelper

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

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

```
class ActionViewHelper extends \Neos\FluidAdaptor\Core\AbstractViewHelper {

  public function initializeArguments() {
    $this->registerArgument('class', 'string', 'CSS class to add to the link');
    $this->registerArgument('target', 'string', 'Target for the link');
    ... and more ...
  }

  public function render() {
    $output = '<a href="..."';
    if ($this->arguments['class']) {
      $output .= ' class="' . $this->arguments['class'] . '"';
    }
    if ($this->arguments['target']) {
      $output .= ' target="' . $this->arguments['target'] . '"';
    }
    $output .= '>';
    ... and more ...
    return $output;
  }

}
```

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

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

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

With the above methods, the `Link\ActionViewHelper` from above can be condensed as follows:

```
class ActionViewHelper extends \Neos\FluidAdaptor\Core\AbstractViewHelper {

    public function initializeArguments() {
        $this->registerUniversalTagAttributes();
    }

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

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

}
```

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

Tip: The `TagBuilder` also makes sure that all attributes are escaped properly, so to decrease the risk of Cross-Site Scripting attacks, make sure to use it when building tags.

---

additionalAttributes

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

We think that the templating framework should not constrain the user in his possibilities – thus, it should be possible to add custom HTML attributes as well, if they are needed. Our solution looks as follows:

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

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

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

This attribute is available in all tags that inherit from `Neos\FluidAdaptor\Core\ViewHelper\AbstractTagBasedViewHelper`.

AbstractConditionViewHelper

To create a custom condition ViewHelper, you need to subclass the `AbstractConditionViewHelper` class, and implement your own static evaluateCondition() method that should return a boolean. The given RenderingContext can provide you with an object manager to get anything you might need to evaluate the condition together with the given arguments.

Depending on the result of this method either the then or the else part is rendered.

@see NeosFluidAdaptorViewHelpersSecurityIfAccessViewHelper::evaluateCondition for a simple usage example.

Every Condition ViewHelper has a "then" and "else" argument, so it can be used like: <[aConditionViewHelperName] .... then="condition true" else="condition false" />, or as well use the "then" and "else" child nodes.

```
class IfAccessViewHelper extends \Neos\FluidAdaptor\Core\ViewHelper\
↪AbstractConditionViewHelper {

/**
 * @param null $arguments
 * @param RenderingContextInterface $renderingContext
 * @return boolean
 */
protected static function evaluateCondition($arguments = null,␣
↪RenderingContextInterface $renderingContext)
{
    $objectManager = $renderingContext->getObjectManager();
    /** @var Context $securityContext */
    $securityContext = $objectManager->get(Context::class);

    if ($securityContext != null && !$securityContext->canBeInitialized()) {
        return false;
    }
    $privilegeManager = static::getPrivilegeManager($renderingContext);
    return $privilegeManager->isPrivilegeTargetGranted($arguments['privilegeTarget'],
↪$arguments['parameters']);
}
```

By basing your condition ViewHelper on the `AbstractConditionViewHelper`, you will get the following features:

- The ViewHelper will have two arguments defined, called `then` and `else`, which are very helpful in the Inline Notation.
- The ViewHelper will automatically work with the `<f:then>` and `<f:else>`-Tags.

Widgets

Widgets are special ViewHelpers which encapsulate complex functionality. It can be best understood what widgets are by giving some examples:

- `<f:widget.paginate>` renders a paginator, i.e. can be used to display large amounts of objects. This is best known from search engine result pages.
- `<f:widget.autocomplete>` adds autocompletion functionality to a text field.
- More widgets could include a Google Maps widget, a sortable grid, ...

Internally, widgets consist of an own Controller and View.

Using Widgets

Using widgets inside your templates is really simple: Just use them like standard ViewHelpers, and consult their documentation for usage examples. An example for the `<f:widget.paginate>` follows below:

```
<f:widget.paginate objects="{blogs}" as="paginatedBlogs" configuration="
↪{itemsPerPage: 10}">
  // use {paginatedBlogs} as you used {blogs} before, most certainly inside
  // a <f:for> loop.
</f:widget.paginate>
```

In the above example, it looks like `{blogs}` contains all `Blog` objects, thus you might wonder if all objects were fetched from the database. However, the blogs are *not fetched* from the database until you actually use them, so the Paginate Widget will adjust the query sent to the database and receive only the small subset of objects.

So, there is no negative performance overhead in using the Paginate Widget.

Writing widgets

We already mentioned that a widget consists of a controller and a view, all triggered by a ViewHelper. We'll now explain these different components one after each other, explaining the API you have available for creating your own widgets.

ViewHelper

All widgets inherit from `Neos\FluidAdaptor\Core\Widget\AbstractWidgetViewHelper`. The ViewHelper of the widget is the main entry point; it controls the widget and sets necessary configuration for the widget.

To implement your own widget, the following things need to be done:

- The controller of the widget needs to be injected into the `$controller` property.
- Inside the `render()`-method, you should call `$this->initiateSubRequest()`, which will initiate a request to the controller which is set in the `$controller` property, and return the `Response` object.
- By default, all ViewHelper arguments are stored as *Widget Configuration*, and are also available inside the Widget Controller. However, to modify the Widget Configuration, you can override the `getWidgetConfiguration()` method and return the configuration which you need there.

There is also a property `$ajaxWidget`, which we will explain later in *Ajax Widgets*.

Controller

A widget contains one controller, which must inherit from `Neos\FluidAdaptor\Core\Widget\AbstractWidgetController`, which is an `ActionController`. There is only one difference between the normal `ActionController` and the `AbstractWidgetController`: There is a property `$widgetConfiguration`, containing the widget's configuration which was set in the ViewHelper.

Fluid Template

The Fluid templates of a widget are normal Fluid templates as you know them, but have a few ViewHelpers available additionally:

<f:uri.widget> Generates an URI to another action of the widget.

<f:link.widget> Generates a link to another action of the widget.

<f:renderChildren> Can be used to render the child nodes of the Widget ViewHelper, possibly with some more variables declared.

Ajax Widgets

Widgets have special support for AJAX functionality. We'll first explain what needs to be done to create an AJAX compatible widget, and then explain it with an example.

To make a widget AJAX-aware, you need to do the following:

- Set `$ajaxWidget` to TRUE inside the ViewHelper. This will generate an unique AJAX Identifier for the Widget, and store the WidgetConfiguration in the user's session on the server.
- Inside the index-action of the Widget Controller, generate the JavaScript which triggers the AJAX functionality. There, you will need a URI which returns the AJAX response. For that, use the following ViewHelper inside the template:

```
<f:uri.widget ajax="TRUE" action="..." arguments="..." />
```

- Inside the template of the AJAX request, `<f:renderChildren>` is not available, because the child nodes of the Widget ViewHelper are not accessible there.

XSD schema generation

A XSD schema file for your ViewHelpers can be created by executing

```
./flow documenation:generatexsd <Your>\\<Package>\\ViewHelpers
    --target-file /some/directory/your.package.xsd
```

Then import the XSD file in your favorite IDE and map it to the namespace `http://typo3.org/ns/<Your/Package>/ViewHelpers`. Add the namespace to your Fluid template by adding the `xmlns` attribute to the root tag (usually `<xml ...>` or `<html ...>`).

Note: You are able to use a different XML namespace pattern by specifying the `--xsd-namespace argument` in the generatexsd command.

If you want to use this inside partials, you can use the "section" argument of the render ViewHelper in order to only render the content of the partial.

Partial:

```
<html xmlns:x="http://typo3.org/ns/Your/Package/ViewHelpers">
<f:section name="content">
    <x:yourViewHelper />
</f:section>
```

Template:

```
<f:render partial="PartialName" section="content" />
```

2.3.10 Validation

Validation in web applications is a very crucial topic: Almost all data which is entered by an end user needs some checking rules, no matter if he enters an e-mail address or a subject for a forum posting.

While validation itself is quite simple, embedding it into the rest of the framework is not: If the user has entered a wrong value, the original page has to be re-displayed, and the user needs some well-readable information on what data he should enter.

This chapter explains:

- how to use the validators being part of Flow
- how to write your own validators
- how to use validation in your own code
- how validation is embedded in the model, the persistence and the MVC layer

Automatic Validation Throughout The Framework

Inside Flow, validation is triggered automatically at two places: When an object is *persisted*, its *base validators* are checked as explained in the last section. Furthermore, validation happens in the MVC layer when a Domain Model is used as a controller argument, directly after Property Mapping.

> **Warning:** If a validation error occurs during persistence, there is no way to catch this error and handle it – as persistence is executed at the end of every request *after the response has been sent to the client*.
>
> Thus, validation on persistence is merely a safeguard for preventing invalid data to be stored in the database.

When validation in the MVC layer happens, it is possible to handle errors correctly. In a nutshell, the process is as follows:

- an array of data is received from the client
- it is transformed to an object using Property Mapping
- this object is validated using the base validators
- if there is a property mapping or validation error, the last page (which usually contains an edit-form) is re-displayed, an error message is shown and the erroneous field is highlighted.

Tip: If you want to suppress the re-display of the last page (which is handled through `errorAction()`, you can add a `@Flow\IgnoreValidation("$comment")` annotation to the docblock of the corresponding controller action.

Normally, you build up your Controller with separate actions for displaying a form to edit an entity and another action to actually create/remove/update the entity. For those actions the validation for Domain Model arguments is triggered as explained above. So in order for the automatic re-display of the previous edit form to work, the validation inside that action needs to be suppressed, or else it would itself possibly fail the validation and try to redirect to previous action, ending up in an infinite loop.

```
class CommentController extends \Neos\Flow\Mvc\Controller\ActionController
{

    /**
     * @param \YourPackage\Domain\Model\Comment $comment
     * @Flow\IgnoreValidation("$comment")
     */
    public function editAction(\YourPackage\Domain\Model\Comment $comment)
    {
        // here, $comment is not necessarily a valid object
    }

    /**
     * @param \YourPackage\Domain\Model\Comment $comment
     */
    public function updateAction(\YourPackage\Domain\Model\Comment $comment)
    {
        // here, $comment is a valid object
    }
}
```

Warning: You should *always* annotate the model arguments of your form displaying actions to ignore validation, or else you might end up with an infinite loop on failing validation.

Furthermore, it is also possible to execute *additional validators* only for specific action arguments using `@Flow\Validate` inside a controller action:

```
class CommentController extends \Neos\Flow\Mvc\Controller\ActionController {

    /**
     * @param \YourPackage\Domain\Model\Comment $comment
     * @Flow\Validate(argumentName="comment", type="YourPackage:SomeSpecialValidator")
     */
    public function updateAction(\YourPackage\Domain\Model\Comment $comment)
    {
        // here, $comment is a valid object
    }
}
```

Tip: It is also possible to add an additional validator for a sub object of the argument, using the "dot-notation": `@Flow\Validate(argumentName="comment.text", type="....")`.

However, it is a rather rare use-case that a validation rule needs to be defined only in the controller.

Using Validators & The ValidatorResolver

A validator is a PHP class being responsible for checking validity of a certain object or simple type.

All validators implement `\Neos\Flow\Validation\Validator\ValidatorInterface`, and the API of every validator is demonstrated in the following code example:

```
// NOTE: you should always use the ValidatorResolver to create new
// validators, as it is demonstrated in the next section.
$validator = new \Neos\Flow\Validation\Validator\StringLengthValidator(array(
    'minimum' => 10,
    'maximum' => 20
));

// $result is of type Neos\Error\Messages\Result
$result = $validator->validate('myExampleString');
$result->hasErrors(); // is FALSE, as the string is longer than 10 characters.

$result = $validator->validate('short');
$result->hasErrors(); // is TRUE, as the string is too short.
$result->getFirstError()->getMessage(); // contains the human-readable error message
```

On the above example, it can be seen that validators can be *re-used* for different input. Furthermore, a validator does not only just return TRUE or FALSE, but instead returns a `Result` object which you can ask whether any errors happened. Please see the API for a detailed description.

Note: The `Neos\Error\Messages\Result` object has been introduced in order to make more structured error output possible – which is especially needed when objects with sub-properties should be validated recursively.

Creating Validator Instances: The ValidatorResolver

As validators can be both singleton or prototype objects (depending if they have internal state), you should not instantiate them directly as it has been done in the above example. Instead, you should use the `\Neos\Flow\Validation\ValidatorResolver` singleton to get a new instance of a certain validator:

```
$validatorResolver->createValidator($validatorType, array $validatorOptions);
```

`$validatorType` can be one of the following:

- a fully-qualified class name to a validator, like `Your\Package\Validation\Validator\FooValidator`
- If you stick to the `<PackageKey>\Validation\Validator\<ValidatorName>Validator` convention, you can also fetch the above validator using `Your.Package:Foo` as `$validatorType`.

  This is the recommended way for custom validators.
- For the standard validators inside the `Neos.Flow` package, you can leave out the package key, so you can use `EmailAddress` to fetch `Neos\Flow\Validation\Validator\EmailAddressValidator`

The `$validatorOptions` parameter is an associative array of validator options. See the validator reference in the appendix for the configuration options of the built-in validators.

Default Validators

Flow is shipped with a big list of validators which are ready to use – see the appendix for the full list. Here, we just want to highlight some more special validators.

Additional to the simple validators for strings, numbers and other basic types, Flow has a few powerful validators shipped:

- `GenericObjectValidator` validates an object by validating all of its properties. This validator is often used internally, but will rarely be used directly.
- `CollectionValidator` validates a collection of objects. This validator is often used internally, but will rarely be used directly.
- `ConjunctionValidator` and `DisjunctionValidator` implement logical AND / OR conditions.

Furthermore, almost all validators of simple types regard `NULL` and the empty string (`''`) as **valid**. The only exception is the `NotEmpty` validator, which disallows both `NULL` and empty string. This means if you want to validate that a property is e.g. an email address *and* does exist, you need to combine the two validators using a `ConjunctionValidator`:

```
$conjunctionValidator = $validatorResolver->createValidator('Conjunction');
$conjunctionValidator->addValidator($validatorResolver->createValidator('NotEmpty'));
$conjunctionValidator->addValidator($validatorResolver->createValidator('EmailAddress
↪'));
```

Validating Domain Models

It is very common that a full Domain Model should be validated instead of only a simple type. To make this use-case more easy, the `ValidatorResolver` has a method `getBaseValidatorConjunction` which returns a fully-configured validator for an arbitrary Domain Object:

```
$commentValidator = $validatorResolver->getBaseValidatorConjunction(
    \YourPackage\Domain\Model\Comment::class, // class name of the object to validate
    ['Default']                               // optional validation groups to use␣
↪during validation
);
$result = $commentValidator->validate($comment);
```

The returned validator checks the following things:

- All *property validation rules* configured through `@Flow\Validate` annotations on properties of the model:

```
namespace YourPackage\Domain\Model;
use Neos\Flow\Annotations as Flow;

class Comment
{

    /**
     * @Flow\Validate(type="NotEmpty")
     */
    protected $text;

    // Add getters and setters here
}
```

It also correctly builds up validators for `Collections` or `arrays`, if they are properly typed (`Doctrine\Common\Collection<YourPackage\Domain\Model\Author>`).

- In addition to validating the individual properties on the model, it checks whether a designated *Domain Model Validator* exists; i.e. for the Domain Model `YourPackage\Domain\Model\Comment` it is checked whether `YourPackage\Domain\Validator\CommentValidator` exists. If it exists, it is automatically called on validation.

  These *Domain Model Validators* can also mark some specific properties as failed and add specific error messages:

```
class CommentValidator extends AbstractValidator
{
    public function isValid($value)
    {
        if ($value instanceof \YourPackage\Domain\Model\Comment) {
            $this->pushResult()->forProperty('text')->addError(
                        new Error('text can´t be empty.', 1221560910)
                    );
        }
    }
}
```

Normally, you would need to annotate Collection and Model type properties, so that the collection elements and the model would be validated like this:

```
/**
 * @var SomeDomainModel
 * @Flow\Validate(type="GenericObject")
 */
protected $someRelatedModel;

/**
 * @var Collection<SomeOtherDomainModel>
 * @Flow\Validate(type="Collection")
 */
protected $someOtherRelatedModels;
```

For convenience, those validators will be added automatically if they are left out, because Flow will always validate Model hierarchies. In some cases, it might be necessary to override validation behaviour of those properties, e.g. when you want to limit validation with Validation Groups (see below). In that case, you can just explicitly annotate the property with additional options and this will then override the automatically generated validator.

When specifying a Domain Model as an argument of a controller action, all the above validations will be automatically executed. This is explained in detail in the following section.

Validation on Aggregates

In Domain Driven Design, the `Aggregate` is to be considered a *consistency boundary*, meaning that the whole `Aggregate` needs to preserve it's invariants at all times. For that reason, validation inside an `Aggregate` will cascade into all entities and force relations to be loaded. So if you have designed large `Aggregates` with a deep hierarchy of many n-ToMany relations, validation can easily become a performance bottleneck.

It is therefore, but not limited to this reason, highly recommended to keep your `Aggregates` small. The validation will stop at an `Aggregate Root`, if the relation to it is lazy and not yet loaded. Entity relations are lazy by default, and as long as you don't also submit parts of the related `Aggregate`, it will not get loaded before the validation kicks in.

Tip: Be careful though, that loading the related Aggregate in your Controller will still make it get validated during persistence. That is another good reason why you should try to minimize relations between Aggregates and if possible, try to stick to a simple identifier instead of an object relation.

For a good read on designing Aggregates, you are highly encouraged to take a read on Vaughn Vernon's essay series Effective Aggregate Design.

Advanced Feature: Partial Validation

If you only want to validate parts of your objects, f.e. want to store incomplete objects in the database, you can assign special *Validation Groups* to your validators.

It is possible to specify a list of validation groups at each `@Flow\Validate` annotation, if none is specified the validation group `Default` is assigned to the validator.

When *invoking* validation, f.e. in the MVC layer or in persistence, only validators with certain validation groups are executed:

- In MVC, the validation group `Default` and `Controller` is used.
- In persistence, the validation group `Default` and `Persistence` is used.

Additionally, it is possible to specify a list of validation groups at each controller action via the `@Flow\ValidationGroups` annotation. This way, you can override the default validation groups that are invoked on this action call, for example when you need to validate uniqueness of a property like an e-mail adress only in your createAction.

A validator is only executed if at least one validation group overlap.

The following example demonstrates this:

```
class Comment
{

    /**
     * @Flow\Validate(type="NotEmpty")
     */
    protected $prop1;

    /**
     * @Flow\Validate(type="NotEmpty", validationGroups={"Default"})
     */
    protected $prop2;

    /**
     * @Flow\Validate(type="NotEmpty", validationGroups={"Persistence"})
     */
    protected $prop3;

    /**
     * @Flow\Validate(type="NotEmpty", validationGroups={"Controller"})
     */
    protected $prop4;

    /**
     * @Flow\Validate(type="NotEmpty", validationGroups={"createAction"})
     */
```

(continues on next page)

(continued from previous page)

```
    protected $prop5;
}

class CommentController extends \Neos\Flow\Mvc\Controller\ActionController
{

    /**
     * @param Comment $comment
     * @Flow\ValidationGroups({"createAction"})
     */
    public function createAction(Comment $comment)
    {
        ...
    }
}
```

- validation for prop1 and prop2 are the same, as the "Default" validation group is added if none is specified
- validation for prop1 and prop2 are executed both on persisting and inside the controller
- validation for $prop3 is only executed in persistence, but not in controller
- validation for $prop4 is only executed in controller, but not in persistence
- validation for $prop5 is only executed in createAction, but not in persistence

If interacting with the `ValidatorResolver` directly, the to-be-used validation groups can be specified as the last argument of `getBaseValidatorConjunction()`.

Note: When trying to set the validation groups of a collection or a whole model, which are normally not annotated for you can explicitly specify a "Collection" or "GenericObject" type validator on the property and set the according validationGroup.

Avoiding Duplicate Validation and Recursion

Unlike simple types, objects (or collections) may reference other objects, potentially leading to recursion during the validation and multiple validation of the same instance.

To avoid this the `GenericObjectValidator` as well as anything extending `AbstractCompositeValidator` keep track of instances that have already been validated. The container to keep track of these instances can be (re-)set using `setValidatedInstancesContainer` defined in the `ObjectValidatorInterface`.

Flow resets this container before doing validation automatically. If you use validation directly in your controller, you should reset the container directly before validation, after any changes have been done.

When implementing your own validators (see below), you need to pass the container around and check instances against it. See `AbstractCompositeValidator` and `isValidatedAlready` in the `GenericObjectValidator` for examples of how to do this.

Another optimization option of the `GenericObjectValidator` is the `skipUnInitializedProxies` flag. When set to true, it allows to skip validation of uninitialized proxy instances, to avoid recursions down into unchanged hierarchies. This can avoid loading of data for validation and is safe, if you can rely on your data not being changed and thus making an entity state invalid "from the outside."

Writing Validators

Usually, when writing your own validator, you will not directly implement `ValidatorInterface`, but rather subclass `AbstractValidator`. You only need to specify any options your validator might use and implement the `isValid()` method then:

```
/**
 * A validator for checking items against foos.
 */
class MySpecialValidator extends \Neos\Flow\Validation\Validator\AbstractValidator
{

    /**
     * @var array
     */
    protected $supportedOptions = array(
        'foo' => array(NULL, 'The foo value to accept as valid', 'mixed', TRUE)
    );

    /**
     * Check if the given value is a valid foo item. What constitutes a valid foo is␣
↪determined through the 'foo' option.
     *
     * @param mixed $value
     * @return void
     */
    protected function isValid($value) {
        if (!isset($this->options['foo'])) {
            throw new \Neos\Flow\Validation\Exception\
↪InvalidValidationOptionsException(
                'The option "foo" for this validator needs to be specified', 12346788
            );
        }

        if ($value !== $this->options['foo']) {
            $this->addError('The value must be equal to "%s"', 435346321, array($this-
↪>options['foo']));
        }
    }
}
```

In the above example, the `isValid()` method has been implemented, and the parameter `$value` is the data we want to check for validity. In case the data is valid, nothing needs to be done.

> **Warning:** You should avoid overwriting `validate()` and if you do, you should never overwrite `$this->result` instance variable of the validator. Instead, use `pushResult()` to create a new result object and at the end of your validator, return `popResult()`.

In case the data is invalid, `$this->addError()` should be used to add an error message, an error code (which should be the unix timestamp of the current time) and optional arguments which are inserted into the error message.

The options of the validator can be accessed in the associative array `$this->options`. The options must be declared as shown above. The $supportedOptions array is indexed by option name and each value is an array with the following numerically indexed elements:

default value of the option # description of the option (used for documentation rendering) # type of the option (used for documentation rendering) # required option flag (optional, defaults to FALSE)

The default values are set in the constructor of the abstract validators provided with Flow. If the required flag is set, missing options will cause an `InvalidValidationOptionsException` to be thrown when the validator is instantiated.

In case you do further checks on the options and any of them is invalid, an `InvalidValidationOptionsException` should be thrown as well.

Tip: Because you extended AbstractValidator in the above example, `NULL` and empty string are automatically regarded as valid values; as it is the case for all other validators. If you do not want to accept empty values, you need to set the class property $acceptsEmptyValues to FALSE.

2.3.11 Property Mapping

The Property Mappers task is to convert *simple types*, like arrays, strings, numbers, to objects. This is most prominently needed in the MVC framework: When a request arrives, it contains all its data as simple types, that is strings, and arrays.

We want to help the developer thinking about *objects*, that's why we try to transparently convert the incoming data to its correct object representation. This is the objective of the *Property Mapper*.

At first, we show some examples on how the property mapper can be used, and then the internal structure is explained.

The main API of the `PropertyMapper` is very simple: It just consists of one method `convert($source, $targetType)`, which receives input data as the first argument, and the target type as second argument. This method returns the built object of type `$targetType`.

Example Usage

The most simple usage is to convert simple types to different simple types, i.e. converting a numeric `string` to a `float` number:

```
// $propertyMapper is of class Neos\Flow\Property\PropertyMapper
$result = $propertyMapper->convert('12.5', 'float');
// $result == (float)12.5
```

This is of course a really conceived example, as the same result could be achieved by just casting the numeric string to a floating point number.

Our next example goes a bit further and shows how we can use the Property Mapper to convert an array of data into a domain model:

```
/**
 * @Flow\Entity
 */
class Neos\MyPackage\Domain\Model\Person {
        /**
         * @var string
         */
        protected $name;

        /**
         * @var \DateTime
         */
        protected $birthDate;
```

(continues on next page)

(continued from previous page)

```
        /**
         * @var Neos\MyPackage\Domain\Model\Person
         */
        protected $mother;
        // ... furthermore contains getters and setters for the above properties.
}

$inputArray = array(
        'name' => 'John Fisher',
        'birthDate' => '1990-11-14T15:32:12+00:00'
);
$person = $propertyMapper->convert($inputArray, \Neos\MyPackage\Domain\Model\
↪Person::class);

// $person is a newly created object of type Neos\MyPackage\Domain\Model\Person
// $person->name == 'John Fisher'
// $person->birthDate is a DateTime object with the correct date set.
```

We'll first use a simple input array:

```
$input = array(
  'name' => 'John Fisher',
  'birthDate' => '1990-11-14T15:32:12+00:00'
);
```

After calling `$propertyMapper->convert($input, \Neos\MyPackage\Domain\Model\Person::class)`, we receive an ew object of type `Person` which has `$name` set to `John Fisher`, and `$birthDate` set to a `DateTime` object of the specified date. You might now wonder how the PropertyMapper knows how to convert `DateTime` objects and `Person` objects? The answer is: It does not know that. However, the PropertyMapper calls specialized *Type Converters* which take care of the actual conversion.

In our example, three type converters are called:

- First, to convert 'John Fisher' to a string (required by the annotation in the domain model), a `StringConverter` is called. This converter simply passes through the input string, without modification.
- Then, a `DateTimeConverter` is called, whose responsibility is to convert the input string into a valid `DateTime` object.
- At the end, the `Person` object still needs to be built. For that, the `PersistentObjectConverter` is responsible. It creates a fresh `Person` object, and sets the `$name` and `$birthDate` properties which were already built using the type converters above.

This example should illustrate that property mapping is a recursive process, and the PropertyMappers task is exactly to orchestrate the different `TypeConverters` needed to build a big, compound object.

The `PersistentObjectConverter` has some more features, as it supports fetching objects from the persistence layer if an identity for the object is given. Both the following inputs will result in the corresponding object to be fetched from the persistence layer:

```
$input = '14d20100-9d70-11e0-aa82-0800200c9a66';
// or:
$input = array(
  '__identity' => '14d20100-9d70-11e0-aa82-0800200c9a66'
);
```

(continues on next page)

(continued from previous page)

```
$person = $propertyMapper->convert($input, 'MyCompany\MyPackage\Domain\Model\Person');
// The $person object with UUID 14d20100-9d70-11e0-aa82-0800200c9a66 is fetched from␣
↪the persistence layer
```

In case some more properties are specified in the array (besides `__identity`), the submitted properties are modified on the fetched object. These modifications are not automatically saved to the database at the end of the request, you need to pass such an instance to `update` on the corresponding repository to persist the changes.

So, let's walk through a more complete input example:

```
$input = array(
  '__identity' => '14d20100-9d70-11e0-aa82-0800200c9a66',
  'name' => 'John Doe',
  'mother' => 'efd3b461-6f24-499d-97bc-309dfbe01f05'
);
```

In this case, the following steps happen:

- The `Person` object with identity `14d20100-9d70-11e0-aa82-0800200c9a66` is fetched from persistence.
- The `$name` of the fetched `$person` object is updated to `John Doe`
- As the `$mother` property is also of type `Person`, the `PersistentObjectConverter` is invoked recursively. It fetches the `Person` object with identifier `efd3b461-6f24-499d-97bc-309dfbe01f05`, which is then set as the `$mother` property of the original person.

Here, you see that we can also set associations using the Property Mapper.

Configuring the Conversion Process

It is possible to configure the conversion process by specifying a `PropertyMappingConfiguration` as third parameter to `PropertyMapper::convert()`. If no `PropertyMappingConfiguration` is specified, the `PropertyMappingConfigurationBuilder` automatically creates a default `PropertyMappingConfiguration`.

In most cases, you should use the `PropertyMappingConfigurationBuilder` to create a new PropertyMappingConfiguration, so that you get a convenient default configuration:

```
        // Here $propertyMappingConfigurationBuilder is an instance of
        // \Neos\Flow\Property\PropertyMappingConfigurationBuilder
$propertyMappingConfiguration = $propertyMappingConfigurationBuilder->build();

        // modify $propertyMappingConfiguration here

        // pass the configuration to convert()
$propertyMapper->convert($source, $targetType, $propertyMappingConfiguration);
```

The following configuration options exist:

- `setMapping($sourcePropertyName, $targetPropertyName)` can be used to rename properties.

  Example: If the input array contains a property `lastName`, but the accordant property in the model is called `$givenName`, the following configuration performs the renaming:

  ```
  $propertyMappingConfiguration->setMapping('lastName', 'givenName');
  ```

- `setTypeConverter($typeConverter)` can be used to directly set a type converter which should be used. This disables the automatic resolving of type converters.
- `setTypeConverterOption($typeConverterClassName, $optionKey, $optionValue)` can be used to set type converter specific options.

  Example: The DateTimeConverter supports a configuration option for the expected date format:

```
$propertyMappingConfiguration->setTypeConverterOption(
        \Neos\Flow\Property\TypeConverter\DateTimeConverter::class,
        \Neos\Flow\Property\TypeConverter\DateTimeConverter::CONFIGURATION_DATE_
↪FORMAT,
        'Y-m-d'
);
```

- `setTypeConverterOptions($typeConverterClassName, array $options)` can be used to set multiple configuration options for the given `TypeConverter`. This overrides all previously set configuration options for the `TypeConverter`.
- `allowProperties($propertyName1, $propertyName2, ...)` specifies the allowed property names which should be converted on the current level.
- `allowAllProperties()` allows *all* properties on the current level.
- `allowAllPropertiesExcept($propertyName1, $propertyName2)` effectively *inverts* the behavior: all properties on the current level are allowed, except the ones specified as arguments to this method.

All the configuration options work only for the current level, i.e. all of the above converter options would only work for the top level type converter. However, it is also possible to specify configuration options for lower levels, using `forProperty($propertyPath)`. This is best shown with the example from the previous section.

The following configuration sets a mapping on the top level, and furthermore configures the `DateTime` converter for the `birthDate` property:

```
$propertyMappingConfiguration->setMapping('fullName', 'name');
$propertyMappingConfiguration
        ->forProperty('birthDate')
        ->setTypeConverterOption(
                \Neos\Flow\Property\TypeConverter\DateTimeConverter::class,
                \Neos\Flow\Property\TypeConverter\DateTimeConverter::CONFIGURATION_
↪DATE_FORMAT,
                'Y-m-d'
        );
```

`forProperty()` also supports more than one nesting level using the dot notation, so writing something like `forProperty('mother.birthDate')` is possible. For multi-valued property types (`Doctrine\Common\Collections\Collection` or `array`) the property mapper will use indexes as property names. To match the property mapping configuration for any index, the path syntax supports an asterisk as a placeholder:

```
$propertyMappingConfiguration
        ->forProperty('items.*')
        ->setTypeConverterOption(
                \Neos\Flow\Property\TypeConverter\PersistentObjectConverter::class,
                \Neos\Flow\Property\TypeConverter\
↪PersistentObjectConverter::CONFIGURATION_CREATION_ALLOWED,
                TRUE
        );
```

This also allows to easily configure TypeConverter options, like for the DateTimeConverter, for subproperties on large collections:

```
$propertyMappingConfiguration
        ->forProperty('persons.*.birthDate')
        ->setTypeConvertOption(
                \Neos\Flow\Property\TypeConverter\DateTimeConverter::class,
                \Neos\Flow\Property\TypeConverter\DateTimeConverter::CONFIGURATION_
↪DATE_FORMAT,
                'Y-m-d'
        );
```

Property Mapping Configuration in the MVC stack

The most common use-case where you will want to adjust the Property Mapping Configuration is inside the MVC stack, where incoming arguments are converted to objects.

If you use Fluid forms, normally no adjustments are needed. However, when programming a web service or an ajax endpoint, you might need to set the `PropertyMappingConfiguration` manually. You can access them using the `\Neos\Flow\Mvc\Controller\Argument` object – and this configuration takes place inside the corresponding `initialize*Action` of the controller, as in the following example:

```
protected function initializeUpdateAction() {
  $commentConfiguration = $this->arguments['comment']->
↪getPropertyMappingConfiguration();
  $commentConfiguration->allowAllProperties();
  $commentConfiguration
    ->setTypeConverterOption(
    \Neos\Flow\Property\TypeConverter\PersistentObjectConverter::class,
    \Neos\Flow\Property\TypeConverter\PersistentObjectConverter::CONFIGURATION_
↪CREATION_ALLOWED,
    TRUE
  );
}

/**
 * @param \My\Package\Domain\Model\Comment $comment
 */
public function updateAction(\My\Package\Domain\Model\Comment $comment) {
  // use $comment object here
}
```

Tip: Maintain IDE's awareness of the `Argument` variable type

Most IDEs will lose information about the variable's type when it comes to array accessing like in the above example `$this->arguments['comment']->...`. In order to keep track of the variables' types, you can synonymously use

```
protected function initializeUpdateAction() {
  $commentConfiguration = $this->arguments->getArgument('comment')->
↪getPropertyMappingConfiguration();
  ...
```

Since the `getArgument()` method is explicitly annotated, common IDEs will recognize the type and there is no break in the type hinting chain.

Mapping whole request body

Sometimes when building an API, you might also want to map the whole request body into a single object, instead of having to only map a single named sub-object. This is often necessary when you don't control the sending side, because you can't expect it to wrap the important request information like this in case of a JSON API:

```
{
  "comment": {
    "author": "john doe",
    "text": "Hello World!"
  }
}
```

Instead you probably receive only the inner object consisting of "author" and "text". For those cases, you can tell Flow that it should map the whole request body into a single action argument with the `@Flow\MapRequestBody("$comment")` annotation on the controller's action method.

```
/**
 * @param \My\Package\Domain\Model\Comment $comment
 * @Flow\MapRequestBody("$comment")
 */
public function createAction(\My\Package\Domain\Model\Comment $comment) {
  // use $comment object here
}
```

Note though, that this will also have the consequence that the comment can no longer be submitted via GET parameters in this action, because the mapping process will directly access the parsed request body and would throw an exception if the body is empty.

Note: Internally, the annotation will only set an attribute on the argument object for the given property name. Hence you can achieve the same without an annotation, by calling `$this->arguments['comment']->setMapRequestBody(true)` inside the `initializeCreateAction()` method.

Security Considerations

The property mapping process can be security-relevant, as a small example should show: Suppose there is a REST API where a person can create a new account, and assign a role to this account (from a pre-defined list). This role controls the access permissions the user has. The data which is sent to the server might look like this:

```
array(
  'username' => 'mynewuser',
  'role' => '5bc42c89-a418-457f-8095-062ace6d22fd'
);
```

Here, the `username` field contains the name of the user, and the `role` field points to the role the user has selected. Now, an attacker could modify the data, and submit the following:

```
array(
  'username' => 'mynewuser',
  'role' => array(
    'name' => 'superuser',
    'admin' => 1
```

(continues on next page)

(continued from previous page)

```
  )
);
```

As the property mapper works recursively, it would create a new `Role` object with the admin flag set to `TRUE`, which might compromise the security in the system.

That's why two parts need to be configured for enabling the recursive behavior: First, you need to specify the allowed properties using one of the `allowProperties()`, `allowAllProperties()` or `allowAllPropertiesExcept()` methods.

Second, you need to configure the the PersistentObjectConverter using the two options `CONFIGURATION_MODIFICATION_ALLOWED` and `CONFIGURATION_CREATION_ALLOWED`. They must be used to explicitly activate the modification or creation of objects. By default, the `PersistentObjectConverter` does only fetch objects from the persistence, but does not create new ones or modifies existing ones.

Note: The only exception to this rule are Value Objects, which may always be created newly by default, as this makes sense as of their nature. If you have a use case where the user may not create new Value Objects, for example because he may only choose from a fixed list, you can however explicitly disallow creation by setting the appropriate property's `CONFIGURATION_CREATION_ALLOWED` option to `FALSE`.

Default Configuration

If the Property Mapper is called without any `PropertyMappingConfiguration`, the `PropertyMappingConfigurationBuilder` supplies a default configuration.

It allows *all changes* for the *top-level object*, but does not allow anything for nested objects.

Note: In the MVC stack, the default `PropertyMappingConfiguration` is much more restrictive, not allowing any changes to any objects. See the next section for an in-depth explanation.

The Common Case: Fluid Forms

The Property Mapper is used to convert incoming values into objects inside the MVC stack.

Most commonly, these incoming values are created using HTML form elements inside Fluid. That is why we want to make sure that only fields which are part of the form are accepted for type conversion, and it should neither be possible to create new objects nor to modify existing ones if that was not intended.

Because of that, the `PropertyMappingConfiguration` inside the MVC stack is configured as restrictive as possible, not allowing any modifications of any objects at all.

Furthermore, Fluid forms render an additional hidden form field containing a secure list of all properties being transmitted; and this list is used to build up the correct `PropertyMappingConfiguration`.

As a result, it is not possible to manipulate the request on the client side, but as long as Fluid forms are used, no extra work has to be done by the developer.

Reference of TypeConverters

Note: This should be automatically generated from the source and will be added to the appendix if available.

The Inner Workings of the Property Mapper

The Property Mapper applies the following steps to convert a simple type to an object. Some of the steps will be described in detail afterwards.

1. Figure out which type converter to use for the given source - target pair.
2. Ask this type converter to return the child properties of the source data (if it has any), by calling `getSourceChildPropertiesToBeConverted()` on the type converter.
3. For each child property, do the following:
    1. Ask the type converter about the data type of the child property, by calling `getTypeOfChildProperty()` on the type converter.
    2. Recursively invoke the `PropertyMapper` to build the child object from the input data.
4. Now, call the type converter again (method `convertFrom()`), passing all (already built) child objects along. The result of this call is returned as the final result of the property mapping process.

On first sight, the steps might seem complex and difficult, but they account for a great deal of flexibility of the property mapper. Automatic resolving of type converters

Automatic Resolving of Type Converters

All type converters which implement `Neos\Flow\Property\TypeConverterInterface` are automatically found in the resolving process. There are four API methods in each `TypeConverter` which influence the resolving process:

`getSupportedSourceTypes()` Returns an array of simple types which are understood as source type by this type converter.

`getSupportedTargetType()` The target type this type converter can convert into. Can be either a simple type, or a class name.

`getPriority()` If two type converters have the same source and target type, precedence is given to the one with higher priority. All standard TypeConverters have a priority lower than 100. A priority of -1 disables automatic resolution for the given TypeConverter!

`canConvertFrom($source, $targetType)` Is called as last check, when source and target types fit together. Here, the TypeConverter can implement runtime constraints to decide whether it can do the conversion.

When a type converter has to be found, the following algorithm is applied:

1. If typeConverter is set in the `PropertyMappingConfiguration`, this is directly used.
2. The inheritance hierarchy of the target type is traversed in reverse order (from most specific to generic) until a TypeConverter is found. If two type converters work on the same class, the one with highest positive priority is used.
3. If no type converter could be found for the direct inheritance hierarchy, it is checked if there is a TypeConverter for one of the interfaces the target class implements. As it is not possible in PHP to order interfaces in any meaningful way, the TypeConverter with the highest priority is used (throughout all interfaces).

4. If no type converter is found in the interfaces, it is checked if there is an applicable type converter for the target type `object`.

If a type converter is found according to the above algorithm, `canConvertFrom` is called on the type converter, so he can perform additional runtime checks. In case the `TypeConverter` returns `FALSE`, the search is continued at the position where it left off in the above algorithm.

For simple target types, the steps 2 and 3 are omitted.

Writing Your Own TypeConverters

Often, it is enough to subclass `Neos\Flow\Property\TypeConverter\AbstractTypeConverter` instead of implementing `TypeConverterInterface`.

Besides, good starting points for own type converters are the `DateTimeConverter` or the `IntegerConverter`. If you write your own type converter, you should set it to a priority greater than 100, to make sure it is used before the standard converters by Flow.

TypeConverters should not contain any internal state, as they are re-used by the property mapper, even recursively during the same run.

Of further importance is the exception and error semantics, so there are a few possibilities what can be returned in `convertFrom()`:

- For fatal errors which hint at some wrong configuration of the developer, throw an exception. This will show a stack trace in development context. Also for detected security breaches, exceptions should be thrown.
- If at run-time the type converter does not wish to participate in the results, `NULL` should be returned. For example, if a file upload is expected, but there was no file uploaded, returning `NULL` would be the appropriate way to handling this.
- If the error is recoverable, and the user should re-submit his data, return a `Neos\Error\Messages\Error` object (or a subclass thereof), containing information about the error. In this case, the property is not mapped at all (`NULL` is returned, like above).

  If the Property Mapping occurs in the context of the MVC stack (as it will be the case in most cases), the error is detected and a forward is done to the last shown form. The end-user experiences the same flow as when MVC validation errors happen.

  This is the correct response for example if the file upload could not be processed because of wrong checksums, or because the disk on the server is full.

Warning: Inside a type converter it is not allowed to use an (injected) instance of `Neos\Flow\Property\PropertyMapper` because it can lead to an infinite recursive invocation.

Note: With version 4.0 TypeConverters with a negative priority will be skipped by the `PropertyMapper` by default. The `PropertyMappingConfiguration` can be used to explicitly use such converter anyways.

2.3.12 Resource Management

Traditionally a PHP application deals directly with all kinds of files. Realizing a file upload is usually an excessive task because you need to create a proper upload form, deal with deciphering the `$_FILES` superglobal and move the uploaded file from the temporary location to a safer place. You also need to analyze the content (is it safe?), control web access and ultimately delete the file when it's not needed anymore.

Flow relieves you of this hassle and lets you deal with simple `PersistentResource` instances instead. File uploads are handled automatically, enforcing the restrictions which were configured by means of validation rules. The publishing mechanism was designed to support a wide range of scenarios, starting from simple publication to the local file system up to fine grained access control and distribution to one or more content delivery networks. This all works without any further ado by you, the application developer.

Storage

The file contents belonging to a specific `PersistentResource` need to be stored in some place, they are not stored in the database together with the object. Applications should be able to store this content in several places as needed, therefor the concept of a *Storage* exists. A *Storage* is configured via `Settings.yaml`:

```
Neos:
  Flow:
    resource:
      storages:
        defaultPersistentResourcesStorage:
          storage: 'Neos\Flow\ResourceManagement\Storage\WritableFileSystemStorage'
          storageOptions:
            path: '%FLOW_PATH_DATA%Persistent/Resources/'
```

The configuration for the `defaultPersistentResourceStorage` (naming for further storages is up to the developer) uses a specific Storage implementation class that abstracts the operations needed for a storage. In this case it is the `WritableFileSystemStorage` which stores data in a given `path` on the local file system of the application. Custom implementations allow you to store their resource contents in other places as needed. You can configure as many storages as you want to separate different types of resources, like your users avatars, generated invoices or any other type of resource you have.

Flow comes configured with two storages by default:

- *defaultStaticResourcesStorage* is the storage for static resources from your packages. This storage is readonly and does not operate on `PersistentResource` instances. See additional information about package resources below.
- *defaultPersistentResourcesStorage* is the general storage for `PersistentResource` content. This storage is used as default if nothing else is specified. Custom storages will most likely be similar to this storage so all of the information below applies.

Target

Flow is a web application framework and as such some (or most) of the resources in the system need to be made accessible online. The resource storages are not meant to be accessible so a `Target` is a configured way of telling how resources are to be published to the web. The default target for our persistent storage above is configured like this:

```
Neos:
  Flow:
    resource:
```

(continues on next page)

(continued from previous page)

```
    targets:
      localWebDirectoryPersistentResourcesTarget:
        target: 'Neos\Flow\ResourceManagement\Target\FileSystemSymlinkTarget'
        targetOptions:
          path: '%FLOW_PATH_WEB%_Resources/Persistent/'
          baseUri: '_Resources/Persistent/'
```

This configures the `Target` named `localWebDirectoryPersistentResourcesTarget`. Resources using this target will be published into the the given `path` which is inside the public web folder of Flow. The class `Neos\Flow\ResourceManagement\Target\FileSystemSymlinkTarget` is the implementation responsible for publishing the resources and providing public URIs to it. From the name you can guess that it creates symlinks to the resources stored on the local filesystem to save space. Other `Target` implementations could publish the resources to CDNs or other external locations that are publicly accessible.

If you have lots of resources in your project you might run into problems when executing `./flow resource:publish` since the number of folders can be limited depending on the file system you're using. An error that might occur in this case is "Could not create directory". To circumvent this error you can tell Flow to split the resources into multiple subfolders in the `_Resources/Persistent` folder of your Web root. The option for your Target you need to set in this case is `subdivideHashPathSegment: TRUE`.

```
Neos:
  Flow:
    resource:
      targets:
        localWebDirectoryPersistentResourcesTarget:
          target: 'Neos\Flow\ResourceManagement\Target\FileSystemSymlinkTarget'
          targetOptions:
            path: '%FLOW_PATH_WEB%_Resources/Persistent/'
            baseUri: '_Resources/Persistent/'
            subdivideHashPathSegment: TRUE
```

Collections

Flow bundles your `PersistentResource``s into collections to allow separation of different types of resources. A ``Collection` is the binding between a `Storage` and a `Target` and each `PersistentResource` belongs to exactly one `Collection` and by that is stored in the matching storage and published to the matching target. You can configure as many collections as you need for specific parts of your application. Flow comes preconfigured with two default collections:

- *static* which is the collection using the `defaultStaticResourcesStorage` and `localWebDirectoryStaticResourcesTarget` to work with (static) package resources. This Collection is meant read-only, which is reflected by the storage used. In this Collection all resources from all packages `Resources/Public/` folders reside.
- *persistent* which is the collection using the `Storage` and `Target` described in the respective section above to store any `PersistentResource` contents by default. Any new `PersistentResource` you create will end up in this storage if not set differently.

Package Resources

Flow packages may provide any amount of static resources. They might be images, stylesheets, javascripts, templates or any other file which is used within the application or published to the web. Static resources may either be public or private:

- *public resources* are represented by the `static Collection` described above and published to a web accessible path.
- *private resources* are not published by default. They can either be used internally (for example as templates) or published with certain access restrictions.

Whether a static package resource is public or private is determined by its parent directory. For a package *Acme.Demo* the public resources reside in a folder called `Acme.Demo/Resources/Public/` while the private resources are stored in `Acme.Demo/Resources/Private/`. The directory structure below *Public* and *Private* is up to you but there are some suggestions in the *chapter about package management*. Both private and public package resources are not represented by `PersistentResource` instances in the database.

Persistent Resources

Data which was uploaded by a user or generated by your application is called a *persistent resource*. Although these resources are usually stored as files, they are never referred to by their path and filename directly but are represented by `PersistentResource` instances.

Note: It is important to completely ignore the fact that resources are stored as files somewhere – you should only deal with resource objects, this allows your application to scale by using remote resource storages.

New persistent resources can be created by either importing or uploading a file. In either case the result is a new `PersistentResource` which can be attached to any other object. As soon as the `PersistentResource` is removed (can happen by cascade operations of related domain objects if you want) the file data is removed too if it is no longer needed by another `PersistentResource`.

Importing Resources

Importing resources is one way to create a new resource object. The `ResourceManager` provides a simple API method for this purpose:

Example: Importing a new resource

```
class ImageController {

        /**
         * @Flow\Inject
         * @var \Neos\Flow\ResourceManagement\ResourceManager
         */
        protected $resourceManager;

        // ... more code here ...

        /**
         * Imports an image
         *
         * @param string $imagePathAndFilename
         * @return void
```

(continues on next page)

(continued from previous page)

```
         */
        public function importImageAction($imagePathAndFilename) {
                $newResource = $this->resourceManager->importResource(
↪$imagePathAndFilename);

                $newImage = new \Acme\Demo\Domain\Model\Image();
                $newImage->setOriginalResource($newResource);

                $this->imageRepository->add($newImage);
        }
}
```

The `ImageController` in our example provides a method to import a new image. Because an image consists of more than just the image file (we need a title, caption, generate a thumbnail, ...) we created a whole new model representing an image. The imported resource is considered as the "original resource" of the image and the `Image` model could easily provide a "thumbnail resource" for a smaller version of the original.

This is what happens in detail while executing the `importImageAction` method:

1. The URI (in our case an absolute path and filename) is passed to the `importResource()` method which analyzes the file found at that location.
2. The file is imported into Flow's persistent resources storage using the sha1 hash over the file content as its filename. If a file with exactly the same content is imported it will reuse the already stored file data.
3. The ResourceManager returns a new `PersistentResource` which refers to the newly imported file.
4. A new `Image` object is created and the resource is attached to it.
5. The image is added to the `ImageRepository` to persist it.

In order to delete a resource just disconnect the resource object from the persisted object, for example by unsetting `originalResource` in the `Image` object and call the `deleteResource()` method in the ResourceManager.

The `importResource()` method also accepts stream resources instead of file URIs to fetch the content from and you can give the name of the resource `Collection` as second argument to define where to store your new resource.

If you already have the new resource`s content available as a string you can use `importResourceFromContent()` to create a resource object from that.

Resource Uploads

The second way to create new resources is uploading them via a POST request. Flow's MVC framework detects incoming file uploads and automatically converts them into `PersistentResource` instances. In order to persist an uploaded resource you only need to persist the resulting object.

Consider the following Fluid template:

```
<f:form method="post" action="create" object="{newImage}" objectName="newImage"
        enctype="multipart/form-data">
        <f:form.textfield property="title" value="My image title" />
        <f:form.upload property="originalResource" />
        <f:form.submit value="Submit new image"/>
</f:form>
```

This form allows for submitting a new image which consists of an image title and the image resource (e.g. a JPEG file). The following controller can handle the submission of the above form:

```
class ImageController {

    /**
     * Creates a new image
     *
     * @param \Acme\Demo\Domain\Model\Image $newImage The new image
     * @return void
     */
    public function createAction(\Acme\Demo\Domain\Model\Image $newImage) {
        $this->imageRepository->add($newImage);
        $this->forward('index');
    }
}
```

Provided that the `Image` class has a `$title` and a `$originalResource` property and that they are accessible through `setTitle()` and `setOriginalResource()` respectively the above code will work just as expected:

```
use Doctrine\ORM\Mapping as ORM;

class Image {

    /**
     * @var string
     */
    protected $title;

    /**
     * @var \Neos\Flow\ResourceManagement\PersistentResource
     * @ORM\OneToOne
     */
    protected $originalResource;

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

    /**
     * @return string
     */
    public function getTitle() {
        return $this->title;
    }

    /**
     * @param \Neos\Flow\ResourceManagement\PersistentResource $originalResource
     * @return void
     */
    public function setOriginalResource(\Neos\Flow\ResourceManagement\
↪PersistentResource $originalResource) {
        $this->originalResource = $originalResource;
    }

    /**
```

(continues on next page)

(continued from previous page)

```
     * @return \Neos\Flow\ResourceManagement\PersistentResource
     */
    public function getOriginalResource() {
        return $this->originalResource;
    }
}
```

All resources are imported into the default *persistent* `Collection` if nothing else was configured. You can either set an alternative collection name in the template.

```
<f:form method="post" action="create" object="{newImage}" objectName="newImage"
        enctype="multipart/form-data">
        <f:form.textfield property="title" value="My image title" />
        <f:form.upload property="originalResource" collection="images" />
        <f:form.submit value="Submit new image"/>
</f:form>
```

Or you can define it in your property mapping configuration like this:

```
$propertyMappingConfiguration
        ->forProperty('originalResource')
        ->setTypeConverterOption(
                \Neos\Flow\ResourceManagement\ResourceTypeConverter::class,
                \Neos\Flow\ResourceManagement\ResourceTypeConverter::CONFIGURATION_
↪COLLECTION_NAME,
                'images'
        );
```

Both variants would import the uploaded resource into a collection named *images*. All import methods in the `ResourceManager` described above allow setting the collection as well.

Tip: If you want to see the internals of file uploads you can check the `ResourceTypeConverter` code.

Accessing Resources

There are multiple ways of accessing your resource`s data depending on what you want to do. Either you need a web accessible URI to a resource to display or link to it or you need the raw data to process it further (like image manipulation for example).

To provide URIs your resources have to be published. For newly created `PersistentResource` objects this happens automatically. Package resources have to be published at least once by running the `resource:publish` command:

```
path$ ./flow resource:publish
```

This will publish all collections, you can also just publish the *static* `Collection` by using the `--collection` argument.

Why Flow uses symbolic links by default

Publishing resources basically means copying files from the `Storage` location to the `Target`. In the default configuration Flow instead creates symbolic links, making the resources consume less disk space and work faster. By changing the `Target` configuration you can change this.

Package Resources

Static resources (provided by packages) need to be published by the `resource:publish` command. If you do not change the default configuration the whole `Resources/Public/` folder is symlinked, which means you probably never need to publish again. If you configure some other `Target` make sure to publish the *static* collection whenever your package resources change.

To get the URI to a published package resource you can use the `getPublicPersistentResourceUri()` method in the `ResourceManager` like this:

```
$resourceUri = $this->resourceManager->getPublicPackageResourceUri('Acme.Demo',
↪'Images/Icons/FooIcon.png');
```

The same can be done in Fluid templates by using the the built-in resource ViewHelper:

```
<img src="{f:uri.resource(path: 'Images/Icons/FooIcon.png', package: 'Acme.Demo')}" />
```

Note that the `package` parameter is optional and defaults to the package containing the currently active controller.

Warning: Although it might be a tempting shortcut, never refer to the resource files directly through a URL like `_Resources/Static/Packages/Acme.Demo/Images/Icons/FooIcon.png` because you can't really rely on this path. Always use the resource view helper instead.

Persistent Resources

Persistent resources are published on creation to the configured `Target`. To get the URI for it you can rely on the `ResourceManager` and use the `getPublicPersistentResourceUri` method with your resource object:

```
$resourceUri = $this->resourceManager->getPublicPersistentResourceUri($image->
↪getOriginalResource());
```

Again in a Fluid template the resource ViewHelper generates the URI for you:

```
<img src="{f:uri.resource(resource: image.originalResource)}" />
```

A persistent resource published to the default `Target` is accessible through a web URI like `http://example.local/_Resources/Persistent/107bed85ba5e9bae0edbae879bbc2c26d72033ab/your_filename.jpg`. One advantage of using the sha1 hash of the resource content as part of the path is that once the resource changes it gets a new path and is displayed correctly regardless of the cache settings in the user's web browser.

If you need to access a resource`s data directly in your code you can aquire a stream via the `getStream()` method of the `PersistentResource`. If a stream is not enough and you need a file path to work with the `createTemporaryLocalCopy()` will return one for you.

Warning: The file in the path returned by `createTemporaryLocalCopy()` is just valid for the current request and also just for reading. You should neither delete nor write to this temporary file. Also don't store this path.

Resource Stream Wrapper

Static resources are often used by packages internally. Typical use cases are templates, XML, YAML or other data files and images for further processing. You might be tempted to refer to these files by using one of the `FLOW_PATH_*` constants or by creating a path relative to your package. A much better and more convenient way is using Flow's built-in package resources stream wrapper.

The following example reads the content of the file `Acme.Demo/Resources/Private/Templates/SomeTemplate.html` into a variable:

Example: Accessing static resources

```
$template = file_get_contents(
        'resource://Acme.Demo/Private/Templates/SomeTemplate.html'
);
```

Some situations might require access to persistent resources. The resource stream wrapper also supports this. To use this feature, just pass the resource hash:

Example: Accessing persisted resources

```
$imageFile = file_get_contents('resource://' . $resource->getSha1());
```

You are encouraged to use this stream wrapper wherever you need to access a static or persistent resource in your PHP code.

Publishing to a Content Delivery Network (CDN)

Flow can publish resources to Content Delivery Networks or other remote services by using specialized connectors.

First you need to install your desired connector (a third-party package which usually can be obtained through packagist.org9 configure it according to its documentation (provide correct credentials etc).

Once the connector package is in place, you add a new publishing target which uses that connect and assign this target to your collection.

```
Neos:
  Flow:
    resource:
      collections:
        persistent:
          target: 'cloudFrontPersistentResourcesTarget'
      targets:
        cloudFrontPersistentResourcesTarget:
          target: 'Flownative\Aws\S3\S3Target'
          targetOptions:
            bucket: 'media.example.com'
            keyPrefix: '/'
            baseUri: 'https://abc123def456.cloudfront.net/'
```

Since the new publishing target will be empty initially, you need to publish your assets to the new target by using the `resource:publish` command:

```
path$ ./flow resource:publish
```

This command will upload your files to the target and use the calculated remote URL for all your assets from now on.

Switching the storage of a collection (move to CDN)

If you want to migrate from your default local filesystem storage to a remote storage, you need to copy all your existing persistent resources to that new storage and use that storage afterwards by default.

You start by adding a new storage with the desired driver that connects the resource management to your CDN. As you might want also want to serve your assets by the remote storage system, you also add a target that contains your published resources (as with local storage this can't be the same as the storage).

```
Neos:
  Flow:
    resource:
      storages:
        s3PersistentResourcesStorage:
          storage: 'Flownative\Aws\S3\S3Storage'
          storageOptions:
            bucket: 'storage.example.com'
            keyPrefix: 'my/assets/'
      targets:
        s3PersistentResourcesTarget:
          target: 'Flownative\Aws\S3\S3Target'
          targetOptions:
            bucket: 'media.example.com'
            keyPrefix: '/'
            baseUri: 'https://abc123def456.cloudfront.net/'
```

In order to copy the resources to the new storage we need a temporary collection that uses the storage and the new publication target.

```
Neos:
  Flow:
    resource:
      collections:
        tmpNewCollection:
          storage: 's3PersistentResourcesStorage'
          target: 's3PersistentResourcesTarget'
```

Now you can use the `resource:copy` command:

```
path$ ./flow resource:copy --publish persistent tmpNewCollection
```

This will copy all your files from your current storage (local filesystem) to the new remote storage. The `--publish` flag means that this command also publishes all the resources to the new target, and you have the same state on your current storage and publication target as on the new one.

Now you can overwrite your old collection configuration and remove the temporary one:

```
Neos:
  Flow:
    resource:
      collections:
        persistent:
          storage: 's3PersistentResourcesStorage'
          target: 's3PersistentResourcesTarget'
```

Clear caches and you're done.

2.3.13 Routing

As explained in the Model View Controller chapter, in Flow the dispatcher passes the request to a controller which then calls the respective action. But how to tell, what controller of what package is the right one for the current request? This is were the Routing Framework comes into play.

The Router

The request builder asks the router for the correct package, controller and action. For this it passes the current request to the routers `route()` method. The router then iterates through all configured routes and invokes their `matches()` method. The first route that matches, determines which action will be called with what parameters.

The same works for the opposite direction: If a link is generated the routers `resolve()` method calls the `resolve()` method of all routes until one route can return the correct URI for the specified arguments.

Note: If no matching route can be found, a `NotFoundException` is thrown which results in a 404 status code for the HTTP response and an error page being displayed. In Development context that error page contains some more details about the error that occurred.

Routes

A route describes the way from your browser to the controller - and back.

With the `uriPattern` you can define how a route is represented in the browser's address bar. By setting `defaults` you can specify package, controller and action that should apply when a request matches the route. Besides you can set arbitrary default values that will be available in your controller. They are called `defaults` because you can overwrite them by so called *dynamic route parts*.

But let's start with an easy example:

Example: Simple route - Routes.yaml

```
-
  name: 'Homepage'
  uriPattern: ''
  defaults:
    '@package': 'My.Demo'
    '@controller': 'Standard'
    '@action': 'index'
```

Note: `name` is optional, but it's recommended to set a name for all routes to make debugging easier.

If you insert these lines at the beginning of the file `Configurations/Routes.yaml`, the `indexAction` of the `StandardController` in your *My.Demo* package will be called when you open up the homepage of your Flow installation (`http://localhost/`).

URI patterns

The URI pattern defines the appearance of the URI. In a simple setup the pattern only consists of *static route parts* and is equal to the actual URI (without protocol and host).

In order to reduce the amount of routes that have to be created, you are allowed to insert markers, so called *dynamic route parts*, that will be replaced by the Routing Framework. You can even mark route parts *optional*.

But first things first.

Static route parts

A static route part is really simple - it will be mapped one-to-one to the resulting URI without transformation.

Let's create a route that calls the `listAction` of the `ProductController` when browsing to `http://localhost/my/demo`:

Example: Simple route with static route parts Configuration/Routes.yaml

```
-
  name: 'Static demo route'
  uriPattern: 'my/demo'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
    '@action':     'list'
```

Dynamic route parts

Dynamic route parts are enclosed in curly brackets and define parts of the URI that are not fixed.

Let's add some dynamics to the previous example:

Example: Simple route with static and dynamic route parts - Configuration/Routes.yaml

```
-
  name: 'Dynamic demo route'
  uriPattern: 'my/demo/{@action}'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
```

Now `http://localhost/my/demo/list` calls the `listAction` just like in the previous example.

With `http://localhost/my/demo/new` you'd invoke the `newAction` and so on.

Note: It's not allowed to have successive dynamic route parts in the URI pattern because it wouldn't be possible to determine the end of the first dynamic route part then.

The `@` prefix should reveal that *action* has a special meaning here. Other predefined keys are `@package`, `@subpackage`, `@controller` and `@format`. But you can use dynamic route parts to set any kind of arguments:

Example: dynamic parameters - Configuration/Routes.yaml

```
-
  name: 'Dynamic demo route with parameter'
  uriPattern: 'products/list/{sortOrder}.{@format}'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
    '@action':     'list'
```

Browsing to `http://localhost/products/list/descending.xml` will then call the `listAction` in your `Product` controller and the request argument `sortOrder` has the value of `descending`.

By default, dynamic route parts match any simple type and convert it to a string that is available through the corresponding request argument. Read on to learn how you can use objects in your routes.

Object Route Parts

If a route part refers to an object, that is *known to the Persistence Manager*, it will be converted to its technical identifier (usually the UUID) automatically:

Example: object parameters - Configuration/Routes.yaml

```
-
  name: 'Single product route'
  uriPattern: 'products/{product}'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
    '@action':     'show'
```

If you add this route *above the previously generated dynamic routes*, an URI pointing to the show action of the ProductController will look like `http://localhost/products/afb275ed-f4a3-49ab-9f2f-1adff12c674f`.

Probably you prefer more human readable URIs and you get them by specifying the `object type`:

```
-
  name: 'Single product route'
  uriPattern: 'products/{product}'
  defaults:
    '@package':     'My.Demo'
    '@controller':  'Product'
    '@action':      'show'
  routeParts:
    product:
      objectType: 'My\Demo\Domain\Model\Product'
```

This will use the *identity* properties of the specified model to generate the URI representation of the product.

Note: If the model contains no identity, the technical identifier is used!

Try adding the `@Flow\Identity` annotation to the name property of the product model. The resulting URI will be `http://localhost/products/the-product-name`

Note: The result will be transliterated, so that it does not contain invalid characters

---

Alternatively you can override the behavior by specifying an `uriPattern` for the object route part:

```
-
  name: 'Single product route'
  uriPattern: 'products/{product}'
  defaults:
    '@package':     'My.Demo'
    '@controller':  'Product'
    '@action':      'show'
  routeParts:
    product:
      objectType: 'My\Demo\Domain\Model\Product'
      uriPattern: '{category.title}/{name}'
```

This will add the title of the product category to the resulting URI: `http://localhost/products/product-category/the-product-name` The route part URI pattern can contain all properties of the object or it's relations.

Note: For properties of type `\DateTime` you can define the date format by appending a PHP date format string separated by colon: `{creationDate:m-Y}`. If no format is specified, the default of `Y-m-d` is used.

Note: If an `uriPattern` is set or the `objectType` contains identity properties, mappings from an object to it's URI representation are stored in the `ObjectPathMappingRepository` in order to make sure that existing links work even after a property has changed! This mapping is not required if no uriPattern is set because in this case the mapping is ubiquitous.

Internally the above is handled by the so called `IdentityRoutePart` that gives you a lot of power and flexibility when working with entities. If you have more specialized requirements or want to use routing for objects that are not known to the Persistence Manager, you can create your custom *route part handlers*, as described below.

Route Part Handlers

Route part handlers are classes that implement `Neos\Flow\Mvc\Routing\DynamicRoutePartInterface`. But for most cases it will be sufficient to extend `Neos\Flow\Mvc\Routing\DynamicRoutePart` and overwrite the methods `matchValue` and `resolveValue`.

Let's have a look at a (very simple) route part handler that allows you to match values against configurable regular expressions:

Example: RegexRoutePartHandler.php

```
class RegexRoutePartHandler extends \Neos\Flow\Mvc\Routing\DynamicRoutePart {

        /**
         * Checks whether the current URI section matches the configured RegEx␣
↪pattern.
         *
         * @param string $requestPath value to match, the string to be checked
         * @return boolean TRUE if value could be matched successfully, otherwise␣
↪FALSE.
         */
        protected function matchValue($requestPath) {
                if (!preg_match($this->options['pattern'], $requestPath, $matches)) {
```

(continues on next page)

(continued from previous page)

```
                        return false;
                }
                $this->value = array_shift($matches);
                return true;
        }

        /**
         * Checks whether the route part matches the configured RegEx pattern.
         *
         * @param string $value The route part (must be a string)
         * @return boolean TRUE if value could be resolved successfully, otherwise␣
↪FALSE.
         */
        protected function resolveValue($value) {
                if (!is_string($value) || !preg_match($this->options['pattern'],
↪$value, $matches)) {
                        return false;
                }
                $this->value = array_shift($matches);
                return true;
        }

}
```

The corresponding route might look like this:

Example: Route with route part handlers Configuration/Routes.yaml

```
-
  name: 'RegEx route - only matches index & list actions'
  uriPattern: 'blogs/{blog}/{@action}'
  defaults:
    '@package':    'My.Blog'
    '@controller': 'Blog'
  routeParts:
    '@action':
      handler:   'My\Blog\RoutePartHandlers\RegexRoutePartHandler'
      options:
        pattern: '/index|list/'
```

The method `matchValue()` is called when translating from an URL to a request argument, and the method `resolveValue()` needs to return an URL segment when being passed a value.

Note: For performance reasons the routing is cached. See *Caching* on how to disable that during development.

Warning: Some examples are missing here, which should explain the API better.

Optional route parts

By putting one or more route parts in round brackets you mark them optional. The following route matches `http://localhost/my/demo` and `http://localhost/my/demo/list.html`.

Example: Route with optional route parts - Configuration/Routes.yaml

```
-
  name: 'Dynamic demo route'
  uriPattern: 'my/demo(/{@action}.html)'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
    '@action':     'list'
```

Note: `http://localhost/my/demo/list` won't match here, because either all optional parts have to match - or none.

Note: You have to define default values for all optional dynamic route parts.

Case Sensitivity

By Default URIs are lower-cased. The following example with a username of "Kasper" will result in `http://localhost/users/kasper`

Example: Route with default case handling

```
-
  uriPattern: 'Users/{username}'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
    '@action':     'show'
```

You can change this behavior for routes and/or dynamic route parts:

Example: Route with customised case handling

```
-
  uriPattern: 'Users/{username}'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
    '@action':     'show'
  toLowerCase: false
  routeParts:
    username:
      toLowerCase: true
```

The option `toLowerCase` will change the default behavior for this route and reset it for the username route part. Given the same username of "Kasper" the resulting URI will now be `http://localhost/Users/kasper` (note the lower case "k" in "kasper").

Note: The predefined route parts `@package`, `@subpackage`, `@controller`, `@action` and `@format` are an exception, they're always lower cased!

Matching of incoming URIs to static route parts is always done case sensitive. So "users/kasper" won't match. For dynamic route parts the case is usually not defined. If you want to handle data coming in through dynamic route parts case-sensitive, you need to handle that in your own code.

Exceeding Arguments

By default arguments that are not part of the configured route values are *not appended* to the resulting URI as *query string*.

If you need this behavior, you have to explicitly enable this by setting `appendExceedingArguments`:

```
-
  uriPattern: 'foo/{dynamic}'
  defaults:
    '@package':    'Acme.Demo'
    '@controller': 'Standard'
    '@action':     'index'
  appendExceedingArguments: true
```

Now route values that are neither defined in the `uriPattern` nor specified in the `defaults` will be appended to the resulting URI: `http://localhost/foo/dynamicValue?someOtherArgument=argumentValue`

This setting is mostly useful for *fallback routes* and it is enabled for the default action route provided with Flow, so that most links will work out of the box.

Note: The setting `appendExceedingArguments` is only relevant for *creating* URIs (resolve). While matching an incoming request to a route, this has no effect. Nevertheless, all query parameters will be available in the resulting action request via `$actionRequest::getArguments()`.

Request Methods

Usually the Routing Framework does not care whether it handles a GET or POST request and just looks at the request path. However in some cases it makes sense to restrict a route to certain HTTP methods. This is especially true for REST APIs where you often need the same URI to invoke different actions depending on the HTTP method.

This can be achieved with a setting `httpMethods`, which accepts an array of HTTP verbs:

```
-
  uriPattern: 'some/path'
  defaults:
    '@package':    'Acme.Demo'
    '@controller': 'Standard'
    '@action':     'action1'
  httpMethods: ['GET']
-
  uriPattern: 'some/path'
  defaults:
    '@package':    'Acme.Demo'
    '@controller': 'Standard'
```

(continues on next page)

(continued from previous page)

```
    '@action':     'action2'
  httpMethods: ['POST', 'PUT']
```

Given the above routes a *GET* request to `http://localhost/some/path` would invoke the `action1Action()` while *POST* and *PUT* requests to the same URI would call `action2Action()`.

Note: The setting `httpMethods` is only relevant for *matching* URIs. While resolving route values to an URI, this setting has no effect.

Subroutes

Flow supports what we call *SubRoutes* enabling you to provide custom routes with your package and reference them in the global routing setup.

Imagine following routes in the `Routes.yaml` file inside your demo package:

Example: Demo Subroutes - My.Demo/Configuration/Routes.yaml

```
-
  name: 'Product routes'
  uriPattern: 'products/{@action}'
  defaults:
    '@controller': 'Product'

-
  name: 'Standard routes'
  uriPattern: '{@action}'
  defaults:
    '@controller': 'Standard'
```

And in your global `Routes.yaml`:

Example: Referencing SubRoutes - Configuration/Routes.yaml

```
-
  name: 'Demo SubRoutes'
  uriPattern: 'demo/<DemoSubroutes>(.{@format})'
  defaults:
    '@package': 'My.Demo'
    '@format':  'html'
  subRoutes:
    'DemoSubroutes':
      package: 'My.Demo'
```

As you can see, you can reference SubRoutes by putting parts of the URI pattern in angle brackets (like `<subRoutes>`). With the subRoutes setting you specify where to load the SubRoutes from.

Instead of adjusting the global `Routes.yaml` you can also include sub routes via `Settings.yaml` - see *Subroutes from Settings*.

Internally the ConfigurationManager merges together the main route with its SubRoutes, resulting in the following routing configuration:

Example: Merged routing configuration

```
-
  name: 'Demo SubRoutes :: Product routes'
  uriPattern: 'demo/products/{@action}.{@format}'
  defaults:
    '@package':    'My.Demo'
    '@format':     'html'
    '@controller': 'Product'

-
  name: 'Demo SubRoutes :: Standard routes'
  uriPattern: 'demo/{@action}.{@format}'
  defaults:
    '@package':    'My.Demo'
    '@format':     'html'
    '@controller': 'Standard'
```

You can even reference multiple SubRoutes from one route - that will create one route for all possible combinations.

Nested Subroutes

By default a SubRoute is loaded from the `Routes.yaml` file of the referred package but it is possible to load SubRoutes from a different file by specifying a `suffix`:

```
-
  name: 'Demo SubRoutes'
  uriPattern: 'demo/<DemoSubroutes>'
  subRoutes:
    'DemoSubroutes':
      package: 'My.Demo'
      suffix:  'Foo'
```

This will load the SubRoutes from a file `Routes.Foo.yaml` in the `My.Demo` package. With that feature you can include multiple Routes with your package (for example providing different URI styles). Furthermore you can nest routes in order to minimize duplication in your configuration. You nest SubRoutes by including different SubRoutes from within a SubRoute, using the same syntax as before. Additionally you can specify a set of `variables` that will be replaced in `name`, `uriPattern` and `defaults` of merged routes:

Imagine the following setup:

global Routes.yaml (`Configuration/Routes.yaml`):

```
-
  name: 'My Package'
  uriPattern: '<MyPackageSubroutes>'
  subRoutes:
    'MyPackageSubroutes':
      package: 'My.Package'
```

default package Routes.yaml (`My.Package/Configuration/Routes.yaml`):

```
-
  name: 'Product'
  uriPattern: 'products/<EntitySubroutes>'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Product'
```

(continues on next page)

(continued from previous page)

```
  subRoutes:
    'EntitySubroutes':
      package: 'My.Package'
      suffix:  'Entity'
      variables:
        'entityName': 'product'

-
  name: 'Category'
  uriPattern: 'categories/<EntitySubroutes>'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Category'
  subRoutes:
    'EntitySubroutes':
      package: 'My.Package'
      suffix:  'Entity'
      variables:
        'entityName': 'category'
```

And in ``My.Package/Configuration/Routes.Entity.yaml``:

```
-
  name: '<entityName> list view'
  uriPattern: ''
  defaults:
    '@action': 'index'

-
  name: '<entityName> detail view'
  uriPattern: '{<entityName>}'
  defaults:
    '@action': 'show'

-
  name: '<entityName> edit view'
  uriPattern: '{<entityName>}/edit'
  defaults:
    '@action': 'edit'
```

This will result in a merged configuration like this:

```
-
  name: 'My Package :: Product :: product list view'
  uriPattern: 'products'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Product'
    '@action':     'index'

-
  name: 'My Package :: Product :: product detail view'
  uriPattern: 'products/{product}'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Product'
    '@action':     'show'
```

(continues on next page)

(continued from previous page)

```
-
  name: 'My Package :: Product :: product edit view'
  uriPattern: 'products/{product}/edit'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Product'
    '@action':     'edit'

-
  name: 'My Package :: Category :: category list view'
  uriPattern: 'categories'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Category'
    '@action':     'index'

-
  name: 'My Package :: Category :: category detail view'
  uriPattern: 'categories/{category}'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Category'
    '@action':     'show'

-
  name: 'My Package :: Category :: category edit view'
  uriPattern: 'categories/{category}/edit'
  defaults:
    '@package':    'My.Package'
    '@controller': 'Category'
    '@action':     'edit'
```

Subroutes from Settings

Having to adjust the main `Routes.yaml` whenever you want to include SubRoutes can be cumbersome and error prone, especially when working with 3rd party packages that come with their own routes. Therefore Flow allows you to include SubRoutes via settings, too:

Settings.yaml (`Configuration/Settings.yaml`):

```
Neos:
  Flow:
    mvc:
      routes:
        'Some.Package': TRUE
```

This will include all routes from the main `Routes.yaml` file of the `Some.Package` (and all its nested SubRoutes if it defines any).

You can also adjust the position of the included SubRoutes:

```
Neos:
  Flow:
    mvc:
```

(continues on next page)

(continued from previous page)

```
      routes:
        'Some.Package':
          position: 'start'
```

Internally Flow uses the `PositionalArraySorter` to resolve the order of SubRoutes loaded from Settings. Following values are supported for the `position` option:

- start (<weight>)
- end (<weight>)
- before <key> (<weight>)
- after <key> (<weight>)
- <numerical-order>

`<weight>` defines the priority in case of conflicting configurations. `<key>` refers to another package key allowing you to set order depending on other SubRoutes.

Note: SubRoutes that are loaded via Settings will always be appended **after** Routes loaded via `Routes.yaml` Therefore you should consider getting rid of the main `Routes.yaml` and only use settings to include routes for greater flexibility.

It's not possible to adjust route defaults or the `UriPattern` when including SubRoutes via Settings, but there are two more options you can use:

```
Neos:
  Flow:
    mvc:
      routes:
        'Some.Package':
          suffix: 'Backend'
          variables:
            'variable1': 'some value'
            'variable2': 'some other value'
```

With `suffix` you can specify a custom filename suffix for the SubRoute. The `variables` option allows you to specify placeholders in the SubRoutes (see *Nested Subroutes*).

Tip: You can use the `flow:routing:list` command to list all routes which are currently active:

```
$ ./flow routing:list

Currently registered routes:
neos/login(/{@action}.{@format})          Neos :: Authentication
neos/logout                               Neos :: Logout
neos/setup(/{@action})                    Neos :: Setup
neos                                      Neos :: Backend Overview
neos/content/{@action}                    Neos :: Backend - Content Module
{node}.html/{type}                        Neos :: Frontend content with format and type
{node}.html                               Neos :: Frontend content with (HTML) format
({node})                                  Neos :: Frontend content without a specified␣
↪format
                                          Neos :: Fallback rule - for when no site has␣
↪been defined yet
```

Route Loading Order and the Flow Application Context

- routes inside more specific contexts are loaded *first*
- and *after* that, global ones, so you can specify context-specific routes

Caching

For performance reasons the routing is cached by default. During development of route part handlers it can be useful to disable the routing cache temporarily. You can do so by using the following configuration in your *Caches.yaml*:

```
Flow_Mvc_Routing_Route:
  backend: Neos\Cache\Backend\NullBackend
Flow_Mvc_Routing_Resolve:
  backend: Neos\Cache\Backend\NullBackend
```

Also it can be handy to be able to flush caches for certain routes programmatically so that they can be regenerated. This is useful for example to update all related routes when an entity was renamed. The `RouterCachingService` allows flushing of all route caches via the `flushCaches()` method. Individual routes can be removed from the cache with the `flushCachesByTag()` method.

Tagging

Any UUID string (see `UuidValidator::PATTERN_MATCH_UUID`) in the route values (when resolving URIs) and the match values (when matching incoming requests) will be added to the cache entries automatically as well as an md5 hash of all URI path segments for matched and resolved routes.

Custom route part handlers can register additional tags to be associated with a route by returning an instance of `MatchResult` / `ResolveResult` instead of `true`/`false`:

Example before: SomePartHandler.php

```
use Neos\Flow\Mvc\Routing\DynamicRoutePart;

class SomePartHandler extends DynamicRoutePart {

        protected function matchValue($requestPath) {
                // custom logic, returning FALSE if the $requestPath doesn't match
                $this->value = $matchedValue;
                return true;
        }

        protected function resolveValue($value) {
                // custom logic, returning FALSE if the $value doesn't resolve
                $this->value = $resolvedPathSegment;
                return true;
        }

}
```

Example now: SomePartHandler.php

```
use Neos\Flow\Mvc\Routing\Dto\MatchResult;
use Neos\Flow\Mvc\Routing\Dto\ResolveResult;
use Neos\Flow\Mvc\Routing\Dto\RouteTags;
use Neos\Flow\Mvc\Routing\DynamicRoutePart;

class SomePartHandler extends DynamicRoutePart {

        protected function matchValue($requestPath) {
                // custom logic, returning FALSE if the $requestPath doesn't match, as before
                return new MatchResult($matchedValue, RouteTags::createFromTag('some-tag'));
        }

        protected function resolveValue($value) {
                // custom logic, returning FALSE if the $value doesn't resolve, as before
                return new ResolveResult($resolvedPathSegment, null, RouteTags::createFromTag('some-tag'));
        }

}
```

All cache entries for routes using the above route part handler will be tagged with *some-tag* and could be flushed with `$routerCachingService->flushCachesByTag('some-tag');`.

URI Constraints

Most route parts only affect the *path* when resolving URIs. Sometimes it can be useful for route parts to affect other parts of the resolved URI. For example there could be routes enforcing *https* URIs, a specific HTTP port or global domain and path pre/suffixes.

In the last code example above the `ResolveResult` was constructed with the second argument being `null`. This argument allows route part handlers to specify `UriConstraints` that can pre-set the following attributes of the resulting URI:

- Scheme (for example “https”)
- Host (for example “www.somedomain.tld”)
- Host prefix (for example “en.”)
- Host suffix (for example “co.uk”)
- Port (for example 443)
- Path (for example “some/path”)
- Path prefix (for example “en/”)
- Path suffix (for example “.html”)

Let’s have a look at another simple route part handler that allows you to enforce https URLs:

Example: HttpsRoutePart.php

```
use Neos\Flow\Mvc\Routing\Dto\ResolveResult;
use Neos\Flow\Mvc\Routing\Dto\UriConstraints;
use Neos\Flow\Mvc\Routing\DynamicRoutePart;
```

(continues on next page)

(continued from previous page)

```
class HttpsRoutePart extends DynamicRoutePart
{
    protected function resolveValue($value)
    {
        return new ResolveResult('', UriConstraints::create()->withScheme('https'));
    }

}
```

If a corresponding route is configured, like:

Example: Routes.yaml

```
-
  name: 'Secure route'
  uriPattern: '{https}'
  defaults:
    '@package':    'My.Demo'
    '@controller': 'Product'
    '@action':     'secure'
  routeParts:
    'https':
      handler: 'My\Demo\HttpsRoutePart'
```

All URIs pointing to the respective action will be forced to be *https://* URIs.

As you can see, in this example the route part handler doesn't affect the URI path at all, so with the configured route this will always point to the homepage. But of course route parts can specify a path (segment) *and* UriConstraints at the same time.

Routing Parameters

The last example only carse about URI *resolving*. What if a route should react to conditions that are not extractable from the request URI path? For example the counter-part to the example above, matching only *https://* URIs?

> **Warning:** One could be tempted to access the current request from within the route part handler using Dependency Injection. But remember that routes are cached and that route part handlers won't be invoked again once a corresponding cache entry exists.

For route part handlers to safely access values that are not encoded in the URI path, those values have to be registered as *Routing Parameters*, usually via a HTTP Component (see respective chapter about *HTTP Foundation*).

A HTTP Component that registers the current request scheme as Routing Parameter could look like this:

Example: HttpsRoutePart.php

```
use Neos\Flow\Http\Component\ComponentContext;
use Neos\Flow\Http\Component\ComponentInterface;
use Neos\Flow\Mvc\Routing\Dto\RouteParameters;
use Neos\Flow\Mvc\Routing\RoutingComponent;

class SchemeRoutingParameterComponent implements ComponentInterface
{
```

(continues on next page)

(continued from previous page)

```
    public function handle(ComponentContext $componentContext)
    {
        $existingParameters = $componentContext->getParameter(RoutingComponent::class,
↪ 'parameters');
        if ($existingParameters === null) {
            $existingParameters = RouteParameters::createEmpty();
        }
        $parameters = $existingParameters->withParameter('scheme', $componentContext->
↪getHttpRequest()->getUri()->getScheme());
        $componentContext->setParameter(RoutingComponent::class, 'parameters',
↪$parameters);
    }
}
```

Now we can extend the `HttpRoutePart` to only match *https://* requests:

Example: HttpsRoutePart.php

```
use Neos\Flow\Mvc\Routing\Dto\ResolveResult;
use Neos\Flow\Mvc\Routing\Dto\UriConstraints;
use Neos\Flow\Mvc\Routing\DynamicRoutePart;

class HttpsRoutePart extends DynamicRoutePart
{
    protected function matchValue($value)
    {
        if ($this->parameters->getValue('scheme') !== 'https') {
      return false;
  }
  return true;
    }

    protected function resolveValue($value)
    {
        return new ResolveResult('', UriConstraints::create()->withScheme('https'));
    }

}
```

Note: For route part handlers to be able to access the *Routing Parameters* they have to implement the `ParameterAwareRoutePartInterface` and its `matchWithParameters()` method. The `DynamicRoutePart` already implements the interface and makes parameters available in the `parameters` field.

2.3.14 Cache Framework

Flow offers a caching framework to cache data. The system offers a wide variety of options and storage solutions for different caching needs. Each cache can be configured individually and can implement its own specific storage strategy.

If configured correctly the caching framework can help to speed up installations, especially in heavy load scenarios. This can be done by moving all caches to a dedicated cache server with specialized cache systems like the Redis key-value store (a.k.a. NoSQL database), or shrinking the needed storage space by enabling compression of data.

Introduction

The caching framework can handle multiple caches with different configurations. A single cache consists of any number of cache entries. A single cache entry is defined by these parts:

identifier A string as unique identifier within this cache. Used to store and retrieve entries.

data The data to be cached.

lifetime A lifetime in seconds of this cache entry. The entry can not be retrieved from cache if lifetime expired.

tags Additional tags (an array of strings) assigned to the entry. Used to remove specific cache entries.

The difference between identifier and tags is hard to understand at first glance, it is illustrated with an example.

About the Identifier

The identifier used to store ("set") and retrieve ("get") entries from the cache holds all information to differentiate entries from each other. For performance reasons, it should be quick to calculate. Suppose there is an resource-intensive extension added as a plugin on two different pages. The calculated content depends on the page on which it is inserted and if a user is logged in or not. So, the plugin creates at maximum four different content outputs, which can be cached in four different cache entries:

- page 1, no user logged in
- page 1, a user is logged in
- page 2, no user logged in
- page 2, a user is logged in

To differentiate all entries from each other, the identifier is build from the page id where the plugin is located, combined with the information whether a user is logged in. These are concatenated and hashed (with `sha1()`, for example). In PHP this could look like this:

```
$identifier = sha1((string)$this->getName() . (string)$this->isUserLoggedIn());
```

When the plugin is accessed, the identifier is calculated early in the program flow. Next, the plugin looks up for a cache entry with this identifier. If there is such an entry, the plugin can return the cached content, else it calculates the content and stores a new cache entry with this identifier. In general the identifier is constructed from all dependencies which specify an unique set of data. The identifier should be based on information which already exist in the system at the point of its calculation. In the above scenario the page id and whether or not a user is logged in are already determined during the frontend bootstrap and can be retrieved from the system quickly.

About Tags

Tags are used to drop specific cache entries if the information an entry is constructed from changes. Suppose the above plugin displays content based on different news entries. If one news entry is changed in the backend, all cache entries which are compiled from this news row must be dropped to ensure that the frontend renders the plugin content again and does not deliver old content on the next frontend call. If for example the plugin uses news number one and two on one page, and news one on another page, the according cache entries should be tagged with these tags:

- page 1, tags news_1, news_2
- page 2, tag news_1

If entry two is changed, a simple backend logic could be created, which drops all cache entries tagged with "news_2", in this case the first entry would be invalidated while the second entry still exists in the cache after the operation. While there is always exactly one identifier for each cache entry, an arbitrary number of tags can be assigned to an entry and

one specific tag can be assigned to mulitple cache entries. All tags a cache entry has are given to the cache when the entry is stored (set).

System Architecture

The caching framework architecture is based on these classes:

Neos\Flow\Cache\CacheFactory Factory class to instantiate caches.

Neos\Flow\Cache\CacheManager Returns the cache frontend of a specific cache. Implements methods to handle cache instances.

Neos\Cache\Frontend\FrontendInterface Interface to handle cache entries of a specific cache. Different frontends exist to handle different data types.

Neos\Cache\Backend\BackendInterface Interface for different storage strategies. A set of implementations exist with different characteristics.

In your code you usually rely on dependency injection to have your caches injected. Thus you deal mainly with the API defined in the `FrontendInterface`.

Configuration

The cache framework is configured in the usual Flow way through YAML files. The most important is *Caches.yaml*, although you may of course use *Objects.yaml* to further configure the way your caches are used. Caches are given a (unique) name and have three keys in their configuration:

frontend The frontend to use for the cache.

backend The backend to use for the cache.

backendOptions The backend options to use.

persistent If the cache should stay persistent.

As an example for such a configuration take a look at the default that is inherited for any cache unless overridden:

Example: Default cache settings

```
##
# Default cache configuration
#
# If no frontend, backend or options are specified for a cache, these values
# will be taken to create the cache.
Default:
  frontend: Neos\Cache\Frontend\VariableFrontend
  backend: Neos\Cache\Backend\FileBackend
  backendOptions:
    defaultLifetime: 0
```

Some backends have mandatory as well as optional parameters (which are documented below). If not all mandatory options are defined, the backend will throw an exception on the first access. To override options for a cache, simply set them in *Caches.yaml* in your global or package *Configuration* directory.

Example: Configuration to use RedisBackend for FooCache

```
FooCache:
  backend: Neos\Cache\Backend\RedisBackend
```

(continues on next page)

(continued from previous page)

```
  backendOptions:
    database: 3
```

Persistent Cache

Caches can be marked as being "persistent" which lets the Cache Manager skip the cache while flushing all other caches or flushing caches by tag. Persistent caches make for a versatile and easy to use low-level key-value-store. Simple data like tokens, preferences or the like which usually would be stored in the file system, can be stored in such a cache. Flow uses a persistent cache for storing an encryption key for the Hash Service and Sessions. The configuration for this cache looks like this:

Example: Persistent cache settings

```
##
# Cache configuration for the HashService
#
# If no frontend, backend or options are specified for a cache, these values
# will be taken to create the cache.
Flow_Security_Cryptography_HashService:
  backend: Neos\Cache\Backend\SimpleFileBackend
  persistent: true
```

Note that, because the cache has been configured as "persistent", the *SimpleFileBackend* will store its data in `Data/Persistent/Cache/Flow_Security_Cryptography_HashService/` instead of using the temporary directory `Data/Temporary/Production/Cache/Flow_Security_Cryptography_HashService/`. You can override the cache directory by specifying it in the cache's backend options.

Application Identifier

The application identifier can be used by cache backends to differentiate cache entries with the same cache identifier in the same storage from each other. For example memcache is global, so if you use it for multiple installations or possibly just for different Flow contexts you need to find a way to separate entries from each other. This setting will do that.

The default of *%FLOW_PATH_ROOT%~%FLOW_APPLICATION_CONTEXT%* is not well suited for installations in which the *FLOW_PATH_ROOT* changes after each deployment, so in such cases you might want to exchange it for some hardcoded value identifying each specific installation:

```
Neos:
  Flow:
    cache:
      applicationIdentifier: 'some-unique-system-identifier'
```

Note: Changing the identifier will make cache entries generated with the old identifier useless.

Cache Frontends

Frontend API

All frontends must implement the API defined in the interface `Neos\Cache\Frontend\FrontendInterface`. All cache operations must be done with these methods.

`getIdentifier()` Returns the cache identifier.

`getBackend()` Returns the backend instance of this cache. It is seldom needed in usual code.

`set()` Sets/overwrites an entry in the cache.

`get()` Return the cache entry for the given identifier.

`getByTag()` Finds and returns all cache entries which are tagged by the specified tag.

`has()` Check for existence of a cache entry.

`remove()` Remove the entry for the given identifier from the cache.

`flush()` Removes all cache entries of this cache.

`flushByTag()` Flush all cache entries which are tagged with the given tag.

`collectGarbage()` Call the garbage collection method of the backend. This is important for backends which are unable to do this internally.

`isValidIdentifier()` Checks if a given identifier is valid.

`isValidTag()` Checks if a given tag is valid.

Check the API documentation for details on these methods.

Available Frontends

Currently three different frontends are implemented, the main difference is the data types which can be stored using a specific frontend.

`Neos\Cache\Frontend\StringFrontend` The string frontend accepts strings as data to be cached.

`Neos\Cache\Frontend\VariableFrontend` Strings, arrays and objects are accepted by this frontend. Data is serialized before it is given to the backend. The igbinary serializer is used transparently (if available in the system) which speeds up the serialization and unserialization and reduces data size. The variable frontend is the most frequently used frontend and handles the widest range of data types. While it can also handle string data, the string frontend should be used in this case to avoid the additional serialization done by the variable frontend.

`Neos\Cache\Frontend\PhpFrontend` This is a special frontend to cache PHP files. It extends the string frontend with the method `requireOnce()` and allows PHP files to be `require()`'d if a cache entry exists.

This can be used to cache and speed up loading of calculated PHP code and becomes handy if a lot of reflection and dynamic PHP class construction is done. A backend to be used with the PHP frontend must implement the

`Neos\Cache\Backend\PhpCapableBackendInterface` Currently the file backend is the only backend which fulfills this requirement.

Note: The PHP frontend can only be used to cache PHP files, it does not work with strings, arrays or objects.

Cache Backends

Currently already a number of different storage backends exists. They have different characteristics and can be used for different caching needs. The best backend depends on given server setup and hardware, as well as cache type and usage. A backend should be chosen wisely, a wrong decision could slow down an installation in the end.

Common Options

Common cache backend options

Options	Description	Mandatory	Type	Default
defaultLifeTime	Default lifetime in seconds of a cache entry if it is not specified for a specific entry on set()	No	integer	3600

Note: The `SimpleFileBackend` does **not support** lifetime for cache entries!

Neos\Cache\Backend\SimpleFileBackend

The simple file backend stores every cache entry as a single file to the file system.

By default, cache entries will be stored in a directory below `Data/Temporary/{context}/Cache/`. For caches which are marked as *persistent*, the default directory is `Data/Persistent/Cache/`. You may override each of the defaults by specifying the `cacheDirectory` backend option (see below).

The simple file backend implements the `PhpCapableInterface` and can be used in combination with the `PhpFrontend`. The backend was specifically adapted to these needs and has low overhead for get and set operations, it scales very well with the number of entries for those operations. This mostly depends on the file lookup performance of the underlying file system in large directories, and most modern file systems use B-trees which can easily handle millions of files without much performance impact.

Note: The `SimpleFileBackend` is called like that, because it does not support lifetime for cache entries! Nor does it support tagging cache entries!

Note: Under heavy load the maximum `set()` performance depends on the maximum write and seek performance of the hard disk. If for example the server system shows lots of I/O wait in top, the file backend has reached this bound. A different storage strategy like RAM disks, battery backed up RAID systems or SSD hard disks might help then.

Note: The SimpleFileBackend and FileBackend are the only cache backends that are capable of storing the `Flow_Object_Classes` Cache.

Options

Simple file cache backend options

Option	Description	Mandatory	Type	Default
cacheDirectory	Full path leading to a custom cache directory. *Example:* • /tmp/my-cache-directory/	No	string	
defaultLifeTime	Cache entry lifetime is **not supported** in this backend. Entries never expire!	No		

Neos\Cache\Backend\FileBackend

The file backend stores every cache entry as a single file to the file system. The lifetime and tags are added after the data part in the same file.

By default, cache entries will be stored in a directory below `Data/Temporary/{context}/Cache/`. For caches which are marked as *persistent*, the default directory is `Data/Persistent/Cache/`. You may override each of the defaults by specifying the `cacheDirectory` backend option (see below).

The file backend implements the `PhpCapableInterface` and can be used in combination with the `PhpFrontend`. The backend was specifically adapted to these needs and has low overhead for get and set operations, it scales very well with the number of entries for those operations. This mostly depends on the file lookup performance of the underlying file system in large directories, and most modern file systems use B-trees which can easily handle millions of files without much performance impact.

A disadvantage is that the performance of `flushByTag()` is bad and scales just O(n). This basically means that with twice the number of entries the file backend needs double time to flush entries which are tagged with a given tag. This practically renders the file backend unusable for content caches. The reason for this design decision in Flow is that the file backend is mainly used as AOP cache, where `flushByTag()` is only used if a PHP file changes. This happens very seldom on production systems, so get and set performance is much more important in this scenario.

Note: The SimpleFileBackend and FileBackend are the only cache backends that are capable of storing the `Flow_Object_Classes` Cache.

Options

File cache backend options

Option	Description	Mandatory	Type	Default
cacheDirectory	Full path leading to a custom cache directory. *Example:* • /tmp/my-cache-directory/	No	string	

Neos\Cache\Backend\PdoBackend

The PDO backend can be used as a native PDO interface to databases which are connected to PHP via PDO. The garbage collection is implemented for this backend and should be called to clean up hard disk space or memory.

Note: The definition for the cache tables can be found in the directory `Neos.Cache/Resources/Private/`.

The maximum size of each cache entry is limited to what a `MEDIUMTEXT` type can hold. When using MySQL/MariaDB that is 16MiB, other databases may have a different limit.

Warning: This backend is php-capable. Nevertheless it cannot be used to store the proxy-classes from the `Flow_Object_Classes` Cache. It can be used for other code-caches like `Fluid_TemplateCache`, `Eel_Expression_Code` or `Flow_Aop_RuntimeExpressions`. This can be usefull in certain situations to avoid file operations on production environments. If you want to use this backend for code-caching make sure that `allow_url_include` is enabled in php.ini

Options

Pdo cache backend options

Option	Description	Mandatory	Type	Default
dataSourceName	Data source name for connecting to the database. *Examples:* • mysql:host=localhost;dbname=test;charset=utf8mb4 • sqlite:/path/to/sqlite.db • sqlite::memory:	Yes	string	
username	Username to use for the database connection	No		
password	Password to use for the database connection.	No		

Neos\Cache\Backend\RedisBackend

Redis is a key-value storage/database. In contrast to memcached, it allows structured values.Data is stored in RAM but it allows persistence to disk and doesn't suffer from the design problems which exist with the memcached backend implementation. The redis backend can be used as an alternative of the database backend for big cache tables and helps to reduce load on database servers this way. The implementation can handle millions of cache entries each with hundreds of tags if the underlying server has enough memory.

Redis is known to be extremely fast but very memory hungry. The implementation is an option for big caches with lots of data because most important operations perform O(1) in proportion to the number of keys. This basically means that the access to an entry in a cache with a million entries is not slower than to a cache with only 10 entries, at least if there is enough memory available to hold the complete set in memory. At the moment only one redis server can be used at a time per cache, but one redis instance can handle multiple caches without performance loss when flushing a single cache.

The garbage collection task should be run once in a while to find and delete old tags.

The implementation is based on the phpredis module, which must be available on the system. It is recommended to build this from the git repository. Currently redis version 2.2 is recommended.

Note: It is important to monitor the redis server and tune its settings to the specific caching needs and hardware capabilities. There are several articles on the net and the redis configuration file contains some important hints on how to speed up the system if it reaches bounds. A full documentation of available options is far beyond this documentation.

Warning: The redis implementation is pretty young and should be considered as experimental. The redis project itself has a very high development speed and it might happen that the Flow implementation changes to adapt to new versions.

Warning: This backend is php-capable. Nevertheless it cannot be used to store the proxy-classes from the `FLOW_Object_Classes` Cache. It can be used for other code-caches like `Fluid_TemplateCache`,

`Eel_Expression_Code` or `Flow_Aop_RuntimeExpressions`. This can be usefull in certain situations to avoid file operations on production environments. If you want to use this backend for code-caching make sure that `allow_url_include` is enabled in php.ini

Options

Redis cache backend options

Option	Description	Mandatory	Type	Default
hostname	IP address or name of redis server to connect to	No	string	127.0.0.1
port	Port of the Redis server.	Yes	integer	6379
database	Number of the database to store entries. Each cache should use its own database, otherwise all caches sharing a database are flushed if the flush operation is issued to one of them. Database numbers 0 and 1 are used and flushed by the core unit tests and should not be used if possible.	No	integer	0
password	Password used to connect to the redis instance if the redis server needs authentication. Warning: The password is sent to the redis server in plain text.	No	string	
compressionLevel	Set gzip compression level to a specific value.	No	integer (0 to 9)	0

Neos\Cache\Backend\MemcachedBackend

Memcached is a simple key/value RAM database which scales across multiple servers. To use this backend, at least one memcache daemon must be reachable, and the PHP module memcache must be loaded. There are two PHP memcache implementations: memcache and memcached, only memcache is currently supported by this backend.

Warning and Design Constraints

Memcached is by design a simple key-value store. Values must be strings and there is no relation between keys. Since the caching framework needs to put some structure in it to store the identifier-data-tags relations, it stores, for each cache entry, an identifier-to-data, an identifier-to-tags and a tag-to-identifiers entry.

This leads to structural problems:

- **If memcache runs out of memory but must store new entries, it will toss *some* other** entry out of the cache (this is called an eviction in memcached speak).
- **If data is shared over multiple memcache servers and some server fails, key/value pairs** on this system will just vanish from cache.

Both cases lead to corrupted caches: If, for example, a tags-to-identifier entry is lost, `dropByTag()` will not be able to find the corresponding identifier-to-data entries which should be removed and they will not be deleted. This results in old data delivered by the cache. Additionally, there is currently no implementation of the garbage collection which

can rebuild cache integrity. It is thus important to monitor a memcached system for evictions and server outages and to clear clear caches if that happens.

Furthermore memcache has no sort of namespacing. To distinguish entries of multiple caches from each other, every entry is prefixed with the cache name. This can lead to very long runtimes if a big cache needs to be flushed, because every entry has to be handled separately and it is not possible to just truncate the whole cache with one call as this would clear the whole memcached data which might even hold non Flow related entries.

Because of the mentioned drawbacks, the memcached backend should be used with care or in situations where cache integrity is not important or if a cache has no need to use tags at all.

Note: The current native debian squeeze package (probably other distributions are affected, too) suffers from PHP memcache bug 16927.

Note: Since memcached has no sort of namespacing and access control, this backend should not be used if other third party systems do have access to the same memcached daemon for security reasons. This is a typical problem in cloud deployments where access to memcache is cheap (but could be read by third parties) and access to databases is expensive.

Warning: This backend is php-capable. Nevertheless it cannot be used to store the proxy-classes from the `FLOW_Object_Classes` Cache. It can be used for other code-caches like `Fluid_TemplateCache`, `Eel_Expression_Code` or `Flow_Aop_RuntimeExpressions`. This can be usefull in certain situations to avoid file operations on production environments. If you want to use this backend for code-caching make sure that `allow_url_include` is enabled in php.ini

Options

Memcached cache backend options

Option	Description	Mandatory	Type	Default
servers	Array of used memcached servers, at least one server must be defined. Each server definition is a string, allowed syntaxes: • **host** TCP connect to host on memcached default port (usually 11211, defined by PHP ini variable memcache.default_port • **host:port** TCP connect to host on port • **tcp://hostname:port** Same as above • **unix:///path/to/memcached.sock** Connect to memcached server using unix sockets	Yes	array	
compression	Enable memcached internal data compression. Can be used to reduce memcached memory consumption but adds additional compression / decompression CPU overhead on the according memcached servers.	No	boolean	FALSE

Neos\Cache\Backend\ApcuBackend

APCu is also known as APC without opcode cache. It can be used to store user data. As main advantage the data can be shared between different PHP processes and requests. All calls are direct memory calls. This makes this backend lightning fast for get() and set() operations. It can be an option for relatively small caches (few dozens of megabytes) which are read and written very often.

The implementation is very similar to the memcached backend implementation and suffers from the same problems if APCu runs out of memory.

Note: It is not advisable to use the APCu backend in shared hosting environments for security reasons: The user cache in APCu is not aware of different virtual hosts. Basically every PHP script which is executed on the system can read and write any data to this shared cache, given data is not encapsulated or namespaced in any way. Only use the APCu backend in environments which are completely under your control and where no third party can read or tamper your data.

Warning: This backend is php-capable. Nevertheless it cannot be used to store the proxy-classes from the `Flow_Object_Classes` Cache. It can be used for other code-caches like `Fluid_TemplateCache`, `Eel_Expression_Code` or `Flow_Aop_RuntimeExpressions`. This can be useful in certain situations to avoid file operations on production environments. If you want to use this backend for code-caching make sure that `allow_url_include` is enabled in php.ini

Options

The APCu backend has no options.

Neos\Cache\Backend\TransientMemoryBackend

The transient memory backend stores data in a local array. It is only valid for one request. This becomes handy if code logic needs to do expensive calculations or must look up identical information from a database over and over again during its execution. In this case it is useful to store the data in an array once and just lookup the entry from the cache for consecutive calls to get rid of the otherwise additional overhead. Since caches are available system wide and shared between core and extensions they can profit from each other if they need the same information.

Since the data is stored directly in memory, this backend is the quickest backend available. The stored data adds to the memory consumed by the PHP process and can hit the `memory_limit` PHP setting.

Options

The transient memory backend has no options.

Neos\Cache\Backend\NullBackend

The null backend is a dummy backend which doesn't store any data and always returns `FALSE` on `get()`.

Options

The null backend has no options.

Neos\Cache\Backend\MultiBackend

This backend accepts several backend configurations to be used in order of appareance as a fallback mechanismn shoudl one of them not be available. If *backendConfigurations* is an empty array this will act just like the NullBackend.

Warning: Due to the nature of this backend as fallback it will swallow all errors on creating and using the sub backends. So configuration errors won't show up. See *debug* option.

Options

Multi cache backend options

Option	Description	Mandatory	Type	Default
setInAllBackends	Should values given to the backend be replicated into all configured and available backends? Generally that is desirable for fallback purposes, but to avoid too much duplication at the cost of performance on fallbacks this can be disabled.	No	bool	true
backendConfigurations	A list of backends to be used in order of appearance. Each entry in that list should have the keys "backend" and "backendOptions" just as a top level backend configuration.	Yes	array	[]
debug	Switch on debug mode which will throw any errors happening in sub backends. Use this in development to make sure everything works as expected.	No	bool	false

Neos\Cache\Backend\TaggableMultiBackend

Technically all the same as the MultiBackend above but implements the TaggableBackendInterface and so supports tagging.

Options are the same as for the MultiBackend.

How to Use the Caching Framework

This section is targeted at developers who want to use caches for arbitrary needs. It is only about proper initialization, not a discussion about identifier, tagging and lifetime decisions that must be taken during development.

Register a Cache

To register a cache it must be configured in *Caches.yaml* of a package:

```
MyPackage_FooCache:
  frontend: Neos\Cache\Frontend\StringFrontend
```

In this case `\Neos\Cache\Frontend\StringFrontend` was chosen, but that depends on individual needs. This setting is usually not changed by users. Any option not given is inherited from the configuration of the "Default" cache. The name (`MyPackage_FooCache` in this case) can be chosen freely, but keep possible name clashes in mind and adopt a meaningful schema.

Retrieve and Use a Cache

Using dependency injection

A cache is usually retrieved through dependency injection, either constructor or setter injection. Which is chosen depends on when you need the cache to be available. Keep in mind that even if you seem to need a cache in the constructor, you could always make use of `initializeObject()`. Here is an example for setter injection matching the configuration given above. First you need to configure the injection in *Objects.yaml*:

```
MyCompany\MyPackage\SomeClass:
  properties:
    fooCache:
      object:
        factoryObjectName: Neos\Flow\Cache\CacheManager
        factoryMethodName: getCache
        arguments:
          1:
            value: MyPackage_FooCache
```

This configures what will be injected into the following setter:

```
/**
 * Sets the foo cache
 *
 * @param \Neos\Cache\Frontend\StringFrontend $cache Cache for foo data
 * @return void
 */
public function setFooCache(\Neos\Cache\Frontend\StringFrontend $cache) {
        $this->fooCache = $cache;
}
```

To make it even simpler you could omit the setter method and annotate the member with the `Inject` annotations. The injected cache is fully initialized, all available frontend operations like `get()`, `set()` and `flushByTag()` can be executed on `$this->fooCache`.

Using the CacheFactory

Of course you can also manually ask the CacheManager (have it injected for your convenience) for a cache:

```
$this->fooCache = $this->cacheManager->getCache('MyPackage_FooCache');
```

2.3.15 Session Handling

Flow has excellent support for working with sessions.

This chapter will explain:

- ... how to store specific data in a session
- ... how to store objects in the session
- ... how to delete sessions

Scope Session

Flow does not only support the `prototype` and `singleton` object scopes, but also the object scope `session`. Objects marked like this basically behave like `singleton` objects which are automatically serialized into the user's session.

As an example, when building a shopping basket, the class could look as follows:

```
/**
 * @Flow\Scope("session")
 */
class ShoppingBasket {

        /**
         * @var array
         */
        protected $items = array();

        /**
         * @param string $item
         * @return void
         * @Flow\Session(autoStart = TRUE)
         */
        public function addItem($item) {
                $this->items[] = $item;
        }

        /**
         * @return array
         */
        public function getItems() {
                return $this->items;
        }
}
```

In the above example

- the object scope is set to `session`, so it behaves like a *user-bound cross-request singleton*. This `ShoppingBasket` can now be injected where it is needed using *Dependency Injection*.

- We only want to start a session when the first element is added to the shopping basket. For this the addItem() method needs to be annotated with `@Flow\Session(autoStart = TRUE)`.

When a user browses the website, the following then happens:

- First, the user's shopping basket is empty, and getItems() returns an empty array. No session exists yet. For each page being requested, the `ShoppingBasket` is newly initialized.
- As soon as the user adds something to the shopping basket, `addItem()` is called. Because this is annotated with `@Flow\Session(autoStart = TRUE)`, a new PHP session is started, and the ShoppingBasket is placed into the session.
- As the user continues to browse the website, the `ShoppingBasket` is being fetched from the user's session (which exists now). Thus, `getItems()` returns the items from the session.

Why is `@Flow\Session(autoStart = TRUE)` necessary?

If Flow did not have this annotation, there would be no way for it to determine when a session must be started. Thus, every user browsing the website would *always* need a session as soon as an object of scope `session` is accessed. This would happens if the `session`-scoped object is still in its initial state.

To be able to use proxies such as Varnish, Flow defers the creation of a session to a point in time when it is really needed – and the developer needs to tell the framework when that point is reached using the above annotation.

The Flow session scope handles persistent objects and dependency injection correctly:

- Objects which are injected via Dependency Injection are removed before serialization and re-injected on deserialization.
- Persistent objects which are unchanged are just stored as a reference and fetched from persistence again on deserialization.
- Persistent objects which are modified are fully stored in the session.

Low-level session handling

It is possible to inject the `Neos\Flow\Session\SessionInterface` and interact with the session on a low level, by using `start()`, `getData()` and `putData()`.

That should rarely be needed, though. Instead of manually serializing objects object into the session, the *session scope* should be used whenever possible.

Session Backends

The session implementation of Flow is written in pure PHP and uses the caching framework as its storage. This allows for storing session data in a variety of backends, including PDO databases, APC and Redis.

The preferred storage backend for the built-in session is defined through a custom Caches.yaml file, placed in a package or the global configuration directory:

```
Flow_Session_Storage:
  backend: Neos\Cache\Backend\RedisBackend
```

The built-in session implementation provides a few more configuration options, related to the session cookie and the automatic garbage collection. Please refer to the Settings.yaml file of the Flow package for a list of all possible options and their respective documentation.

Session Storage

Note: Since Flow 5.2 Sessions are no longer destroyed by default when caches are flushed. This is due to the session caches being marked as "persistent". This previously lead to all sessions being destroyed on each deployment, which was undesireable.

If you deploy changes that change the structure of data that is stored in the session or the class of an *@Flows-cope("session")* object, you need to manually destroy sessions to avoid deserialization errors. You can do this by running *./flow flow:session:destroyAll* or manually deleting the folder where the sessions are stored.

Also, sessions are shared among the application contexts, e.g. *Development* and *Production*, so if your use case requires to have sessions separated for different contexts, you need to configure the *cacheDirectory* backend option for the *Flow_Session_Storage* and *Flow_Session_MetaData* caches for each individual context. Please refer to the *Cache Framework* section of this guide for further information.

2.3.16 Command Line

Flow features a clean and powerful interface for the command line which allows for automated and manual execution of low-level or application-specific tasks. The command line support is available on all platforms generally supported by Flow.

This chapter describes how to use the help system, how to run existing commands and how to implement your own custom commands.

Wrapper Script

Flow uses two platform specific wrapper scripts for running the actual commands:

- *flow.bat* is used on Windows machines
- *flow* is used on all other platforms

Both files are located and must be run from the main directory of the Flow installation. The command and further options are passed as arguments to the respective wrapper script.

In the following examples we refer to these wrapper scripts just as "the *flow* script".

Tip: If you are a Windows user and use a shell like msysGit, you can mostly follow the Unix style examples and use the *flow* script instead of *flow.bat*.

Help System

Without specifying a command, the *flow* script responds by displaying the current version number and the current context:

```
$ ./flow
Flow 2.x.x ("Development" context)
usage: ./flow <command identifier>

See "./flow help" for a list of all available commands.
```

In addition to the packages delivered with the Flow core, third-party packages may provide any number of custom commands. A list of all currently available commands can be obtained with the *help* command:

```
$ ./flow help
Flow 2.x.x ("Development" context)
usage: ./flow <command identifier>

The following commands are currently available:

PACKAGE "Neos.Flow":
------------------------------------------------------------------------------
* flow:cache:flush                          Flush all caches
  cache:warmup                              Warm up caches

  configuration:show                        Show the active configuration
                                            settings
  configuration:validate                    Validate the given configuration
  configuration:generateschema              Generate a schema for the given
                                            configuration or YAML file.
...
```

A list of all commands in a specific package can be obtained by giving the package key part of the command to the *help* command:

```
$ ./flow help kickstart
5 commands match the command identifier "neos.kickstart":

PACKAGE "Neos.KICKSTART":
------------------------------------------------------------------------------
kickstart:package                       Kickstart a new package
kickstart:actioncontroller              Kickstart a new action controller
kickstart:commandcontroller             Kickstart a new command controller
kickstart:model                         Kickstart a new domain model
kickstart:repository                    Kickstart a new domain repository
```

Further details about specific commands are available by specifying the respective command identifier:

```
$ ./flow help configuration:show


Show the active configuration settings

COMMAND:
  neos.flow:configuration:show

USAGE:
  ./flow configuration:show [<options>]

OPTIONS:
  --type               Configuration type to show
  --path               path to subconfiguration separated by "." like
                       "Neos.Flow

DESCRIPTION:
  The command shows the configuration of the current context as it is used by Flow␣
↪itself.
  You can specify the configuration type and path if you want to show parts of the␣
↪configuration.
```

(continues on next page)

(continued from previous page)

```
./flow configuration:show --type Settings --path Neos.Flow.persistence
```

Running a Command

Commands are uniquely identified by their *command identifier*. These come in two variants: a long and a short version.

Fully Qualified Command Identifier

A fully qualified command identifier is the combination of the package key, the command controller name and the actual command name, separated by colons:

The command "warmup" implemented by the "CacheCommandController" contained in the package "Neos.Flow" is referred to by the command identifier *neos.flow:cache:warmup*.

Short Command Identifier

In order to save some typing, most commands can be referred to by a shortened command identifier. The *help* command lists all commands by the shortest possible identifier which is still unique across all available commands.

For example, the command "warmup" implemented by the "CacheCommandController" contained in the package "Neos.Flow" can also be referred to by the command identifier *cache:warmup* as long as no other package provides a command with the same name.

Some special commands can only by referred to by their fully qualified identifier because they are invoked at a very early stage when the command resolution mechanism is not yet available. These *Compile Time Commands* are marked by an asterisk in the list of available commands (see *Symfony/Console Methods* for some background information).

Passing Arguments

Arguments and options can be specified for a command in the same manner they are passed to typical Unix-like commands. A list of required arguments and further options can be retrieved through the *help* command.

Options

Options listed for a command are optional and only have to be specified if needed. Options must always be passed before any arguments by using their respective name:

```
./flow foo:bar --some-option BAZ --some-argument QUUX
```

If an option expects a boolean type (that is, yes/no, true/false, on/off would be typical states), just specifying the option name is sufficient to set the option to *true*:

```
./flow foo:bar --force
```

Alternatively the boolean value can be specified explicitly:

```
./flow foo:bar --force TRUE
./flow foo:bar --force FALSE
```

Possible values equivalent to *TRUE* are: *on*, *1*, *y*, *yes*, *true*. Possible values equivalent to *FALSE* are: *off*, *0*, *n*, *no*, *false*.

Arguments

The arguments listed for a command are mandatory. They can either be specified by their name or without an argument name. If the argument name is omitted, the argument values must be provided in the same order like in the help screen of the respective command. The following two command lines are synonymic:

```
./flow kickstart:actioncontroller --force --package-key Foo.Bar --controller-name Baz
./flow kickstart:actioncontroller --force Foo.Bar Baz
```

Contexts

If not configured differently by the server environment, the *flow* script is run in the *Development* context by default. It is recommended to set the *FLOW_CONTEXT* environment variable to *Production* on a production server – that way you don't execute commands in an unintended context accidentally.

If you usually run the *flow* script in one context but need to call it in another context occasionally, you can do so by temporarily setting the respective environment variable for the single command run:

```
FLOW_CONTEXT=Production ./flow flow:cache:flush
```

In a Windows shell, you need to use the *SET* command:

```
SET FLOW_CONTEXT=Production
flow.bat flow:cache:flush
```

Implementing Custom Commands

A lot of effort has been made to make the implementation of custom commands a breeze. Instead of writing configuration which registers commands or coming up with files which provide the help screens, creating a new command is only a matter of writing a simple PHP method.

A set of commands is bundled in a *Command Controller*. The individual commands are plain PHP methods with a name that ends with the word "Command". The concrete command controller must be located in a "Command" namespace right below the package's namespace.

The following example illustrates all the code necessary to introduce a new command:

```
namespace Acme\Demo\Command;
use Neos\Flow\Annotations as Flow;

/**
 * @Flow\Scope("singleton")
 */
class CoffeeCommandController extends \Neos\Flow\Cli\CommandController {

        /**
         * Brew some coffee
         *
         * This command brews the specified type and amount of coffee.
         *
         * Make sure to specify a type which best suits the kind of drink
         * you're aiming for. Some types are better suited for a Latte, while
         * others make a perfect Espresso.
         *
         * @param string $type The type of coffee
```

(continues on next page)

(continued from previous page)

```
         * @param integer $shots The number of shots
         * @param boolean $ristretto Make this coffee a ristretto
         * @return string
         */
        public function brewCommand($type, $shots=1, $ristretto=FALSE) {
                # implementation
        }
}
```

The new controller and its command is detected automatically and the help screen is rendered by using the information provided by the method code and DocComment:

- the first line of the DocComment contains the short description of the command
- the second line must be empty
- the the following lines contain the long description
- the descriptions of the @param annotations are used for the argument descriptions
- the type specified in the @param annotations is used for validation and to determine if the argument is a flag (boolean) or not
- the parameters declared in the method set the parameter names and tell if they are arguments (mandatory) or options (optional). All arguments must be placed in front of the options.

The above example will result in a help screen similar to this:

```
$ ./flow help coffee:brew

Brew some coffee

COMMAND:
  acme.demo:coffee:brew

USAGE:
  ./flow coffee:brew

DESCRIPTION:
  This command brews the specified type and amount of coffee.

  Make sure to specify a type which best suits the kind of drink
  you're aiming for. Some types are better suited for a Latte, while
  others make a perfect Espresso.
```

Handling Exceeding Arguments

Any arguments which are passed additionally to the mandatory arguments are considered to be *exceeding arguments*. These arguments are not parsed nor validated by Flow.

A command may use exceeding arguments in order to process an variable amount of parameters. The exceeding arguments can be retrieved through the *Request* object as in the following example:

```
/**
 * Process words
 *
 * This command processes the given words.
```

(continues on next page)

(continued from previous page)

```
     *
     * @param string $operation The operation to execute
     * @return string
     */
    public function processWordCommand($operation = 'uppercase') {
            $words = $this->request->getExceedingArguments();
            foreach ($words as $word) {
                    ...
            }
            ...
    }
```

A typical usage of the command above may look like this:

```
$ ./flow foo:processword --operation lowercase These Are The Words

these are the words
```

See Other and Deprecated Commands

A command's help screen can contain additional information about relations to other commands. This information is triggered by specifying one or more *@see* annotations in the command's doc comment block as follows:

```
/**
 * Drink juice
 *
 * This command provides some way of drinking juice.
 *
 * @return string
 * @see acme.demo:drink:coffee
 */
public function juiceCommand() {
        ...
}
```

By adding a *@deprecated* annotation, the respective command will be marked as deprecated in all help screens and a warning will be displayed when executing the command. If a *@see* annotation is specified, the deprecation message additionally suggests to use the command mentioned there.

```
/**
 * Drink tea
 *
 * This command urges you to drink tea.
 *
 * @return string
 * @deprecated since 2.8.18
 * @see acme.demo:drink:coffee
 */
public function teaCommand() {
        ...
}
```

Generating Styled Output

The output sent to the user can be processed in three different ways, each denoted by a PHP constant:

- OUTPUTFORMAT_RAW sends the output as is
- OUTPUTFORMAT_PLAIN tries to convert the output into plain text by stripping possible tags
- OUTPUTFORMAT_STYLED sends the output as is but converts certain tags into ANSI codes

The output format can be set by calling the *setOutputFormat()* method on the command controller's *Response* object:

```
/**
 * Example Command
 *
 * @return string
 */
public function exampleCommand() {
        $this->response->setOutputFormat(Response::OUTPUTFORMAT_RAW);
        $this->response->appendContent(...);
}
```

A limited number of tags are supported for brushing up the output in OUTPUTFORMAT_STYLED mode. They have the following meaning:

Tag	Meaning
`<b>...</b>`	Render the text in a bold / bright style
`<i>...</i>`	Render the text in a italics
`<u>...</u>`	Underline the given text
`<em>...</em>`	Emphasize the text, usually by inverting foreground and background colors
`<strike>...</strike>`	Display the text struck through

The respective styles are only rendered correctly if the console supports ANSI styles. You can check ANSI support by calling the response's *hasColorSupport()* method. Contrary to what that method name suggests, at the time of this writing colored output is not directly supported by Flow. However, such a feature is planned for the future.

Tip: The tags supported by Flow can also be used to style the description of a command in its DocComment.

Symfony/Console Methods

The CommandController makes use of Symfony/Console internally and provides various methods directly from the CommandController's `output` member:

- TableHelper
  - outputTable($rows, $headers = NULL)
- DialogHelper
  - select($question, $choices, $default = NULL, $multiSelect = false, $attempts = FALSE)
  - ask($question, $default = NULL, array $autocomplete = array())
  - askConfirmation($question, $default = TRUE)
  - askHiddenResponse($question, $fallback = TRUE)

  - askAndValidate($question, $validator, $attempts = FALSE, $default = NULL, array $autocomplete = NULL)
  - askHiddenResponseAndValidate($question, $validator, $attempts = FALSE, $fallback = TRUE)
- ProgressHelper
  - progressStart($max = NULL)
  - progressSet($current)
  - progressAdvance($step = 1)
  - progressFinish()

Here's an example showing of some of those functions:

```
namespace Acme\Demo\Command;

use Neos\Flow\Annotations as Flow;
use Neos\Flow\Cli\CommandController;

/**
 * @Flow\Scope("singleton")
 */
class MyCommandController extends CommandController {

        /**
         * @return string
         */
        public function myCommand() {
                // render a table
                $this->output->outputTable(array(
                        array('Bob', 34, 'm'),
                        array('Sally', 21, 'f'),
                        array('Blake', 56, 'm')
                ),
                array('Name', 'Age', 'Gender'));

                // select
                $colors = array('red', 'blue', 'yellow');
                $selectedColorIndex = $this->output->select('Please select one color',
↪ $colors, 'red');
                $this->outputLine('You choose the color %s.', array($colors[
↪$selectedColorIndex]));

                // ask
                $name = $this->output->ask('What is your name?' . PHP_EOL, 'Bob',␣
↪array('Bob', 'Sally', 'Blake'));
                $this->outputLine('Hello %s.', array($name));

                // prompt
                $likesDogs = $this->output->askConfirmation('Do you like dogs?');
                if ($likesDogs) {
                        $this->outputLine('You do like dogs!');
                }

                // progress
                $this->output->progressStart(600);
                for ($i = 0; $i < 300; $i ++) {
```

(continues on next page)

(continued from previous page)

```
                        $this->output->progressAdvance();
                        usleep(5000);
                }
                $this->output->progressFinish();

        }
}
```

Runtime and Compile Time

The majority of the commands are run at point when Flow is fully initialized and all of the framework features are available. However, for certain low-level operations it is desirable to execute code much earlier in the boot process – during *compile time*. Commands like *neos.flow:cache:flush* or the internal compilation commands which render the PHP proxy classes cannot rely on a fully initialized system.

It is possible – also for custom commands – to run commands run during compile time. The developer implementing such a command must have a good understanding of the inner workings of the bootstrap and parts of the proxy building, because compile time has several limitations, including but not limited to the following:

- dependency injection does not support property injection
- aspects are not yet active
- persistence is not yet enabled
- certain caches have not been built yet

In general, all functionality which relies on proxy classes will not be available during compile time.

If you are sure that compile time is the right choice for your command, you can register it as a compile time command by running an API method in the *boot()* method of your package's *Package* class:

```
namespace Acme\Foo;
use Neos\Flow\Package\Package as BasePackage;

/**
 * Acme.Foo Package
 */
class Package extends BasePackage {

        /**
         * Invokes custom PHP code directly after the package manager has been␣
↪initialized.
         *
         * @param \Neos\Flow\Core\Bootstrap $bootstrap The current bootstrap
         * @return void
         */
        public function boot(\Neos\Flow\Core\Bootstrap $bootstrap) {
                $bootstrap->registerRequestHandler(new \Acme\Foo\Command\
↪MyCommandController($bootstrap));
        }
}
```

For more details you are encouraged to study the implementation of Flow's own compile time commands.

Executing Sub Commands

Most command methods are designed to be called exclusively through the command line and should not be invoked internally through a PHP method call. They may rely on a certain application state, some exceeding arguments provided through the *Request* object or simply are compile time commands which must not be run from runtime commands. Therefore, the safest way to let a command execute a second command is through a PHP sub process.

The PHP bootstrap mechanism provides a method for executing arbitrary commands through a sub process. This method is located in the *Scripts* class and can be used as follows:

```
use Neos\Flow\Annotations as Flow;
use Neos\Flow\Core\Booting\Scripts;

/**
 * @Flow\InjectConfiguration(package="Neos.Flow")
 * @var array
 */
protected $flowSettings;

public function runCommand() {
        $success = Scripts::executeCommand('acme.foo:bar:baz', $this->flowSettings);
}
```

Sometimes it can be useful to execute commands *asynchronously*, for example when triggering time-consuming tasks where the result is not instantly required (like sending confirmation emails, converting files, ...). This can be done with the `Scripts::executeCommandAsync()*` method:

```
public function runCommand() {
        $commandArguments = ['some-argument' => 'some value'];
        Scripts::executeCommandAsync('acme.foo:bar:baz', $this->flowSettings,
↪$commandArguments);
}
```

Note: Because asynchronous commands are invoked in a separate thread, potential exceptions or failures will *not* be reported. While this can be desired, it might require additional monitoring on the command-side (e.g. a failure log).

Quitting and Exit Code

Commands should not use PHP's *exit()* or *die()* method but rather let Flow's bootstrap perform a clean shutdown of the framework. The base *CommandController* provides two API methods for initiating a shutdown and optionally passing an exit code to the console:

- *quit($exitCode)* stops execution right after this command, performs a clean shutdown of Flow.
- *sendAndExit($exitCode)* sends any output buffered in the *Response* object and exits immediately, without shutting down Flow.

The *quit()* method is the recommended way to exit Flow. The other command, *sendAndExit()*, is reserved for special cases where Flow is not stable enough to continue even with the shutdown procedure. An example for such a case is the *neos.flow:cache:flush* command which removes all cache entries which requires an immediate exit because Flow relies on caches being intact.

2.3.17 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm which complements Object-Oriented Programming (OOP) by separating *concerns* of a software application to improve modularization. The separation of concerns (SoC) aims for making a software easier to maintain by grouping features and behavior into manageable parts which all have a specific purpose and business to take care of.

OOP already allows for modularizing concerns into distinct methods, classes and packages. However, some concerns are difficult to place as they cross the boundaries of classes and even packages. One example for such a *cross-cutting concern* is security: Although the main purpose of a Forum package is to display and manage posts of a forum, it has to implement some kind of security to assert that only moderators can approve or delete posts. And many more packages need a similar functionality for protect the creation, deletion and update of records. AOP enables you to move the security (or any other) aspect into its own package and leave the other objects with clear responsibilities, probably not implementing any security themselves.

Aspect-Oriented Programming has been around in other programming languages for quite some time now and sophisticated solutions taking advantage of AOP exist. Flow's AOP framework allows you to use of the most popular AOP techniques in your own PHP application. In contrast to other approaches it doesn't require any special PHP extensions or manual compile steps – and it's a breeze to configure.

Tip: In case you are unsure about some terms used in this introduction or later in this chapter, it's a good idea looking them up (for example at Wikipedia). Don't think that you're the only one who has never heard of a *Pointcut* or *SoC*[1] – we had a hard time learning these too. However, it's worth the hassle, as a common vocabulary improves the communication between developers a lot.

How AOP can help you

Let's imagine you want to log a message inside methods of your domain model:

Example: Logging without AOP:

```
namespace Examples\Forum\Domain\Model;

class Forum {

        /**
         * @Flow\Inject
         * @var \Examples\Forum\Logger\ApplicationLoggerInterface
         */
        protected $applicationLogger;

        /**
         * Delete a forum post and log operation
         *
         * @param \Examples\Forum\Domain\Model\Post $post
         * @return void
         */
        public function deletePost(Post $post) {
                $this->applicationLogger->log('Removing post ' . $post->getTitle(),␣
↪LOG_INFO);
                $this->posts->remove($post);
        }
```

(continues on next page)

[1] SoC could, by the way, also mean "Self-organized criticality" or "Service-oriented Computing" or refer to Google's "Summer of Code" ...

(continued from previous page)

```
}
```

If you have to do this in a lot of places, the logging would become a part of you domain model logic. You would have to inject all the logging dependencies in your models. Since logging is nothing that a domain model should care about, this is an example of a non-functional requirement and a so-called cross-cutting concern.

With AOP, the code inside your model would know nothing about logging. It will just concentrate on the business logic.

Example: Logging with AOP (your class):

```
namespace Examples\Forum\Domain\Model;

class Forum {

        /**
         * Delete a forum post
         *
         * @param \Examples\Forum\Domain\Model\Post $post
         * @return void
         */
        public function deletePost(Post $post) {
                $this->posts->remove($post);
        }

}
```

The logging is now done from an AOP *aspect*. It's just a class tagged with `@aspect` and a method that implements the specific action, an *before advice*. The expression after the `@before` tag tells the AOP framework to which method calls this action should be applied. It's called *pointcut expression* and has many possibilities, even for complex scenarios.

Example: Logging with AOP (aspect):

```
namespace Examples\Forum\Logging;

/**
 * @Flow\Aspect
 */
class LoggingAspect {

        /**
         * @Flow\Inject
         * @var \Examples\Forum\Logger\ApplicationLoggerInterface
         */
        protected $applicationLogger;

        /**
         * Log a message if a post is deleted
         *
         * @param \Neos\Flow\AOP\JoinPointInterface $joinPoint
         * @Flow\Before("method(Examples\Forum\Domain\Model\Forum->deletePost())")
         * @return void
         */
        public function logDeletePost(\Neos\Flow\AOP\JoinPointInterface $joinPoint) {
                $post = $joinPoint->getMethodArgument('post');
```

(continues on next page)

(continued from previous page)

```
                $this->applicationLogger->log('Removing post ' . $post->getTitle(),␣
↪LOG_INFO);
        }

}
```

As you can see the advice has full access to the actual method call, the *join point*, with information about the class, the method and method arguments.

AOP concepts and terminology

At the first (and the second, third, ...) glance, the terms used in the AOP context are not really intuitive. But, similar to most of the other AOP frameworks, we better stick to them, to keep a common language between developers. Here they are:

Aspect An aspect is the part of the application which cross-cuts the core concerns of multiple objects. In Flow, aspects are implemented as regular classes which are tagged by the `@aspect` annotation. The methods of an aspect class represent advices, the properties may be used for introductions.

Join point A join point is a point in the flow of a program. Examples are the execution of a method or the throw of an exception. In Flow, join points are represented by the `Neos\Flow\AOP\JoinPoint` object which contains more information about the circumstances like name of the called method, the passed arguments or type of the exception thrown. A join point is an event which occurs during the program flow, not a definition which defines that point.

Advice An advice is the action taken by an aspect at a particular join point. Advices are implemented as methods of the aspect class. These methods are executed before and / or after the join point is reached.

Pointcut The pointcut defines a set of join points which need to be matched before running an advice. The pointcut is configured by a *pointcut expression* which defines when and where an advice should be executed. Flow uses methods in an aspect class as anchors for pointcut declarations.

Pointcut expression A pointcut expression is the condition under which a join point should match. It may, for example, define that join points only match on the execution of a (target-) method with a certain name. Pointcut expressions are used in pointcut- and advice declarations.

Target A class or method being adviced by one or more aspects is referred to as a target class /-method.

Introduction An introduction redeclares the target class to implement an additional interface. By declaring an introduction it is possible to introduce new interfaces and an implementation of the required methods without touching the code of the original class. Additionally introductions can be used to add new properties to a target class.

The following terms are related to advices:

Before advice A before advice is executed before the target method is being called, but cannot prevent the target method from being executed.

After returning advice An after returning advice is executed after returning from the target method. The result of the target method invocation is available to the after returning advice, but it can't change it. If the target method throws an exception, the after returning advice is not executed.

After throwing advice An after throwing advice is only executed if the target method throwed an exception. The after throwing advice may fetch the exception type from the join point object.

After advice An after advice is executed after the target method has been called, no matter if an exception was thrown or not.

Around advice An around advice is wrapped around the execution of the target method. It may execute code before and after the invocation of the target method and may ultimately prevent the original method from being executed at all. An around advice is also responsible for calling other around advices at the same join point and returning either the original or a modified result for the target method.

Advice chain If more than one around advice exists for a join point, they are called in an onion-like advice chain: The first around advice probably executes some before-code, then calls the second around advice which calls the target method. The target method returns a result which can be modified by the second around advice, is returned to the first around advice which finally returns the result to the initiator of the method call. Any around advice may decide to proceed or break the chain and modify results if necessary.

Flow AOP concepts

Aspect-Oriented Programming was, of course, not invented by us[2]. Since the initial release of the concept, dozens of implementations for various programming languages evolved. Although a few PHP-based AOP frameworks do exist, they followed concepts which did not match the goals of Flow (to provide a powerful, yet developer-friendly solution) when the development of Neos began. We therefore decided to create a sophisticated but pragmatic implementation which adopts the concepts of AOP but takes PHP's specialties and the requirements of typical Flow applications into account. In a few cases this even lead to new features or simplifications because they were easier to implement in PHP compared to Java.

Flow pragmatically implements a reduced subset of AOP, which satisfies most needs of web applications. The join point model allows for intercepting method executions but provides no special support for advising field access[3]. Pointcut expressions are based on well-known regular expressions instead of requiring the knowledge of a dedicated expression language. Pointcut filters and join point types are modularized and can be extended if more advanced requirements should arise in the future.

Implementation overview

Flow's AOP framework does not require a pre-processor or an aspect-aware PHP interpreter to weave in advices. It is implemented and based on pure PHP and doesn't need any specific PHP extension. However, it does require the Object Manager to fulfill its task.

Flow uses PHP's reflection capabilities to analyze declarations of aspects, pointcuts and advices and implements method interceptors as a dynamic proxy. In accordance to the GoF patterns[4], the proxy classes act as a placeholders for the target object. They are true subclasses of the original and override adviced methods by implementing an interceptor method. The proxy classes are generated automatically by the AOP framework and cached for further use. If a class has been adviced by some aspect, the Object Manager will only deliver instances of the proxy class instead of the original.

The approach of storing generated proxy classes in files provides the whole advantage of dynamic weaving with a minimum performance hit. Debugging of proxied classes is still easy as they truly exist in real files.

[2] AOP was rather invented by Gregor Kiczalesand his team at the Xerox Palo Alto Research Center. The original implementation was called AspectJ and is an extension to Java. It still serves as a de-facto standard and is now maintained by the Eclipse Foundation.

[3] Intercepting setting and retrieval of properties can easily be achieved by declaring a before-, after- or around advice.

[4] GoF means Gang of Four and refers to the authors of the classic book *Design Patterns – Elements of Reusable Object-Oriented Software*

Aspects

Aspects are abstract containers which accommodate pointcut-, introduction- and advice declarations. In most frameworks, including Flow, aspects are defined as plain classes which are tagged (annotated) as an aspect. The following example shows the definition of a hypothetical `FooSecurity` aspect:

Example: Declaration of an aspect:

```
namespace Example\MySecurityPackage;

/**
 * An aspect implementing security for Foo
 *
 * @Flow\Aspect
 */
class FooSecurityAspect {

}
```

As you can see, `\Example\MySecurityPackage\FooSecurityAspect` is just a regular PHP class which may (actually must) contain methods and properties. What makes it an aspect is solely the `Aspect` annotation mentioned in the class comment. The AOP framework recognizes this tag and registers the class as an aspect.

Note: A void aspect class doesn't make any sense and if you try to run the above example, the AOP framework will throw an exception complaining that no advice, introduction or pointcut has been defined.

Note: With Flow 4.0+ classes that are marked `final` can now be targeted by AOP advices by default. This can be explicitly disabled with a `@Flow\Proxy(false)` annotation on the class in question.

Pointcuts

If we want to add security to foo, we need a method which carries out the security checks and a definition where and when this method should be executed. The method is an advice which we're going to declare in a later section, the "where and when" is defined by a pointcut expression in a pointcut declaration.

You can either define the pointcut in the advice declaration or set up named pointcuts to help clarify their use.

A named pointcut is represented by a method of an aspect class. It contains two pieces of information: The pointcut name, defined by the method name, and the pointcut expression, declared by an annotation. The following pointcut will match the execution of methods whose name starts with "delete", no matter in which class they are defined:

Example: Declaration of a named pointcut:

```
/**
 * A pointcut which matches all methods whose name starts with "delete".
 *
 * @Flow\Pointcut("method(.*->delete.*())")
 */
public function deleteMethods() {}
```

Pointcut expressions

As already mentioned, the pointcut expression configures the filters which are used to match against join points. It is comparable to an if condition in PHP: Only if the whole condition evaluates to TRUE, the statement is executed - otherwise it will be just ignored. If a pointcut expression evaluates to TRUE, the pointcut matches and advices which refer to this pointcut become active.

Note: The AOP framework AspectJ provides a complete pointcut language with dozens of pointcut types and expression constructs. Flow makes do with only a small subset of that language, which we think already suffice for even complex enterprise applications. If you're interested in the original feature set, it doesn't hurt throwing a glance at the AspectJ Programming Guide.

Pointcut designators

A pointcut expression always consists of two parts: The poincut designator and its parameter(s). The following designators are supported by Flow:

method()

The `method()` designator matches on the execution of methods with a certain name. The parameter specifies the class and method name, regular expressions can be used for more flexibility[5]. It follows the following scheme:

```
method([public|protected] ClassName->methodName())
```

Specifying the visibility modifier (public or protected) is optional - if none is specified, both visibilities will match. The class- and method name can be specified as a regular expression.

Warning: It is not possible to match for *interfaces* within the `method()` pointcut expression. Instead of `method(InterfaceName->methodName())`, use `within(InterfaceName) && method(.*->methodName())`.

Here are some examples for matching method executions:

Example: method() pointcut designator

Matches all public methods in class `Example\MyPackage\MyObject`:

```
method(public Example\MyPackage\MyObject->.*())
```

Matches all methods prefixed with "delete" (even protected ones) in any class of the package `Example.MyPackage`:

```
method(Example\MyPackage.*->delete.*())
```

Matches all methods except injectors in class `Example\MyPackage\MyObject`:

```
method(Example\MyPackage\MyObject->(?!inject).*())
```

---

[5] Internally, PHP's `preg_match()` function is used to match the method name. The regular expression will be enclosed by /^...$/ (without the dots of course). Backslashes will be escaped to make namespace use possible without further hassle.

Note: In other AOP frameworks, including AspectJ™ and Spring™, the method designator does not exist. They rather use a more fine grained approach with designators such as execution, call and cflow. As Flow only supports matching to method execution join points anyway, we decided to simplify things by allowing only a more general method designator.

The `method()` designator also supports so called runtime evaluations, meaning you can specify values for the method's arguments. If those argument values do not match the advice won't be executed. The following example should give you an idea how this works:

Example: Runtime evaluations for the method() pointcut designator

```
method(Example\MyPackage\MyClass->update(title == "Flow", override == TRUE))
```

Besides the method arguments you can also access the properties of the current object or a global object like the party that is currently authenticated. A detailed description of the runtime evaluations possibilities is described below in the section about the `evaluate()` pointcut designator.

class()

The `class()` designator matches on the execution of methods defined in a class with a certain name. The parameter specifies the class name, again regular expressions are allowed here. The `class()` designator follows this simple scheme:

```
class(classname)
```

Example: class() pointcut designator

Matches all methods in class `Example\MyPackage\MyObject`:

```
class(Example\MyPackage\MyObject)
```

Matches all methods in namespace "Service":

```
class(Example\MyPackage\Service\.*)
```

Warning: The `class` pointcut expression does not match interfaces. If you want to match interfaces, use `within()` instead.

within()

The `within()` designator matches on the execution of methods defined in a class of a certain type. A type matches if the class is a subclass of or implements an interface of the given name. The `within()` designator has this simple syntax:

```
within(type)
```

Example: within() pointcut designator

Matches all methods in classes which implement the logger interface:

```
within(Example\Flow\Log\LoggerInterface)
```

Matches all methods in classes which are part of the Foo layer:

```
within(Example\Flow\FooLayerInterface)
```

Note: `within()` will not match on specific nesting in the call stack, even when the name might imply this. It's just a more generic class designator matching whole type hierarchies.

classAnnotatedWith()

The `classAnnotatedWith()` designator matches on classes which are tagged with a certain annotation. Currently only the actual annotation class name can be matched, arguments of the annotation cannot be specified:

```
classAnnotatedWith(annotation)
```

Example: classAnnotatedWith() pointcut designator

Matches all classes which are tagged with Flow's `Entity` annotation:

```
classAnnotatedWith(Neos\Flow\Annotations\Entity)
```

Matches all classes which are tagged with a custom annotation:

```
classAnnotatedWith(Acme\Demo\Annotations\Important)
```

methodAnnotatedWith()

The `methodAnnotatedWith()` designator matches on methods which are annotated with a certain annotation. Currently only the actual annotation class name can be matched, arguments of the annotation cannot be specified. The syntax of this designator is as follows:

```
methodAnnotatedWith(annotation)
```

Example: methodAnnotatedWith() pointcut designator

Matches all method which are annotated with a `Special` annotation:

```
methodAnnotatedWith(Acme\Demo\Annotations\Special)
```

setting()

The setting() designator matches if the given configuration option is set to TRUE, or if an optional given comparison value equals to its configured value. This is helpful to make advices configurable and switch them off in a specific Flow context or just for testing. You can use this designator as follows:

Example: setting() pointcut designator

Matches if "my.configuration.option" is set to TRUE in the current execution context:

```
setting(my.configuration.option)
```

Matches if "my.configuration.option" is equal to "AOP is cool" in the current execution context: (Note: single and double quotes are allowed)

```
setting(my.configuration.option = 'AOP is cool')
```

---

evaluate()

The `evaluate()` designator is used to execute advices depending on constraints that have to be evaluated during runtime. This could be a specific value for a method argument (see the `method()` designator) or checking a certain property of the current object or accessing a global object like the currently authenticated party. In general you can access object properties by the `.` syntax and global objects are registered under the `current.` keyword. Here is an example showing the possibilities:

Example: evaluate() pointcut designator

Matches if the property name of the global party object (the currently authenticated user of the security framework) is equal to "Andi":

```
evaluate(current.userService.currentUser.name == "Andi")
```

Matches if the property someProperty of someObject which is a property of the current object (the object the advice will be executed in) is equal to the name of the currently authenticated user:

```
evaluate(this.someObject.someProperty == current.userService.currentUser.name)
```

Matches if the property someProperty of the current object is equal to one of the values TRUE, "someString" or the address of the currently authenticated user:

```
evaluate(this.someProperty in (TRUE, "someString", current.userService.
currentUser.address))
```

Matches if the accounts array in the current party object contains the account stored in the myAccount property of the current object:

```
evaluate(current.userService.currentUser.accounts contains this.myAccount)
```

Matches if at least one of the entries in the first array exists in the second one:

```
evaluate(current.userService.currentUser.accounts matches ('Administrator',
'Customer', 'User'))
```

```
evaluate(current.userService.currentUser.accounts matches this.accounts)
```

Tip: If you like you can enter more than one constraint in a single evaluate pointcut designator by separating them with a comma. The evaluate designator will only match, if all its conditions evaluated to TRUE.

Note: It is possible to register arbitrary singletons to be available as global objects with the Flow configuration setting `Neos.Flow.aop.globalObjects`.

filter()

If the built-in filters don't suit your needs you can even define your own custom filters. All you need to do is create a class implementing the `Neos\Flow\AOP\Pointcut\PointcutFilterInterface` and develop your own logic for the `matches()` method. The custom filter can then be invoked by using the `filter()` designator:

```
filter(CustomFilterObjectName)
```

Example: filter() pointcut designator

If the current method matches is determined by the custom filter:

```
filter(Example\MyPackage\MyCustomPointcutFilter)
```

Combining pointcut expressions

All pointcut expressions mentioned in previous sections can be combined into a whole expression, just like you may combine parts to an overall condition in an if construct. The supported operators are "&&", "||" and "!" and they have the same meaning as in PHP. Nesting expressions with parentheses is not supported but you may refer to other pointcuts by specifying their full name (i.e. class- and method name). This final example shows how to combine and reuse pointcuts and ultimately build a hierarchy of pointcuts which can be used conveniently in advice declarations:

Example: Combining pointcut expressions:

```
namespace Example\TestPackage;

/**
 * Fixture class for testing pointcut definitions
 *
 * @Flow\Aspect
 */
class PointcutTestingAspect {

        /**
         * Pointcut which includes all method executions in
         * PointcutTestingTargetClasses except those from Target
         * Class number 3.
         *
         * @Flow\Pointcut("method(Example\TestPackage\PointcutTestingTargetClass.*->.
↪*()) && !method(Example\TestPackage\PointcutTestingTargetClass3->.*())")
         */
        public function pointcutTestingTargetClasses() {}
```

(continues on next page)

(continued from previous page)

```
        /**
         * Pointcut which consists of only the
         * Example\TestPackage\OtherPointcutTestingTargetClass.
         *
         * @Flow\Pointcut("method(Example\TestPackage\OtherPointcutTestingTargetClass-
↪>.*())")
         */
        public function otherPointcutTestingTargetClass() {}

        /**
         * A combination of both above pointcuts
         *
         * @Flow\Pointcut("Example\TestPackage\PointcutTestingAspect->
↪pointcutTestingTargetClasses || Example\TestPackage\PointcutTestingAspect->
↪otherPointcutTestingTargetClass")
         */
        public function bothPointcuts() {}

        /**
         * A pointcut which matches all classes from the service layer
         *
         * @Flow\Pointcut("within(Example\Flow\ServiceLayerInterface)")
         */
        public function serviceLayerClasses() {}

        /**
         * A pointcut which matches any method from the BasicClass and all classes
         * from the service layer
         *
         * @Flow\Pointcut("method(Example\TestPackage\Basic.*->.*()) || within(Neos\
↪Flow\Service.*)")
         */
        public function basicClassOrServiceLayerClasses() {}
}
```

Declaring advice

With the aspect and pointcuts in place we are now ready to declare the advice. Remember that an advice is the actual action, the implementation of the concern you want to weave in to some target. Advices are implemented as interceptors which may run before and / or after the target method is called. Four advice types allow for these different kinds of interception: Before, After returning, After throwing and Around.

Other than being of a certain type, advices always come with a pointcut expression which defines the set of join points the advice applies for. The pointcut expression may, as we have seen earlier, refer to other named pointcuts.

Before advice

A before advice allows for executing code before the target method is invoked. However, the advice cannot prevent the target method from being executed, nor can it take influence on other before advices at the same join point.

Example: Declaration of a before advice:

```
/**
 * Before advice which is invoked before any method call within the News
 * package
 *
 * @Flow\Before("class(Example\News\.*->.*())")
 */
public function myBeforeAdvice(\Neos\Flow\AOP\JoinPointInterface $joinPoint) {
}
```

After returning advice

The after returning advice becomes active after the target method normally returns from execution (i.e. it doesn't throw an exception). After returning advices may read the result of the target method, but can't modify it.

Example: Declaration of an after returning advice:

```
/**
 * After returning advice
 *
 * @Flow\AfterReturning("method(public Example\News\FeedAgregator->[import|update].
↪*()) || Example\MyPackage\MyAspect->someOtherPointcut")
 */
public function myAfterReturningAdvice(\Neos\Flow\AOP\JoinPointInterface $joinPoint) {
}
```

After throwing advice

Similar to the "after returning" advice, the after throwing advice is invoked after method execution, but only if an exception was thrown.

Example: Declaration of an after throwing advice:

```
/**
 * After throwing advice
 *
 * @Flow\AfterThrowing("within(Example\News\ImportantLayer)")
 */
public function myAfterThrowingAdvice(\Neos\Flow\AOP\JoinPointInterface $joinPoint) {
}
```

After advice

The after advice is a combination of "after returning" and "after throwing": These advices become active after method execution, no matter if an exception was thrown or not.

Example: Declaration of an after advice:

```
/**
 * After advice
 *
 * @Flow\After("Example\MyPackage\MyAspect->justAPointcut")
 */
public function myAfterAdvice(\Neos\Flow\AOP\JoinPointInterface $joinPoint) {
}
```

Around advice

Finally, the around advice takes total control over the target method and intercepts it completely. It may decide to call the original method or not and even modify the result of the target method or return a completely different one. Obviously the around advice is the most powerful and should only be used if the concern can't be implemented with the alternative advice types. You might already guess how an around advice is declared:

Example: Declaration of an around advice:

```
/**
 * Around advice
 *
 * @Flow\Around("Example\MyPackage\MyAspect->justAPointcut")
 */
public function myAroundAdvice(\Neos\Flow\AOP\JoinPointInterface $joinPoint) {
}
```

Implementing advice

The final step after declaring aspects, pointcuts and advices is to fill the advices with life. The implementation of an advice is located in the same method it has been declared. In that regard, an aspect class behaves like any other object in Flow – you therefore can take advantage of dependency injection in case you need other objects to fulfill the task of your advice.

Accessing join points

As you have seen in the previous section, advice methods always expect an argument of the type `Neos\Flow\AOP\JoinPointInterface`. This join point object contains all important information about the current join point. Methods like getClassName() or getMethodArguments() let the advice method classify the current context and enable you to implement advices in a way that they can be reused in different situations. For a full description of the join point object refer to the API documentation.

Advice chains

Around advices are a special advice type in that they have the power to completely intercept the target method. For any other advice type, the advice methods are called by the proxy class one after another. In case of the around advice, the methods form a chain where each link is responsible to pass over control to the next.

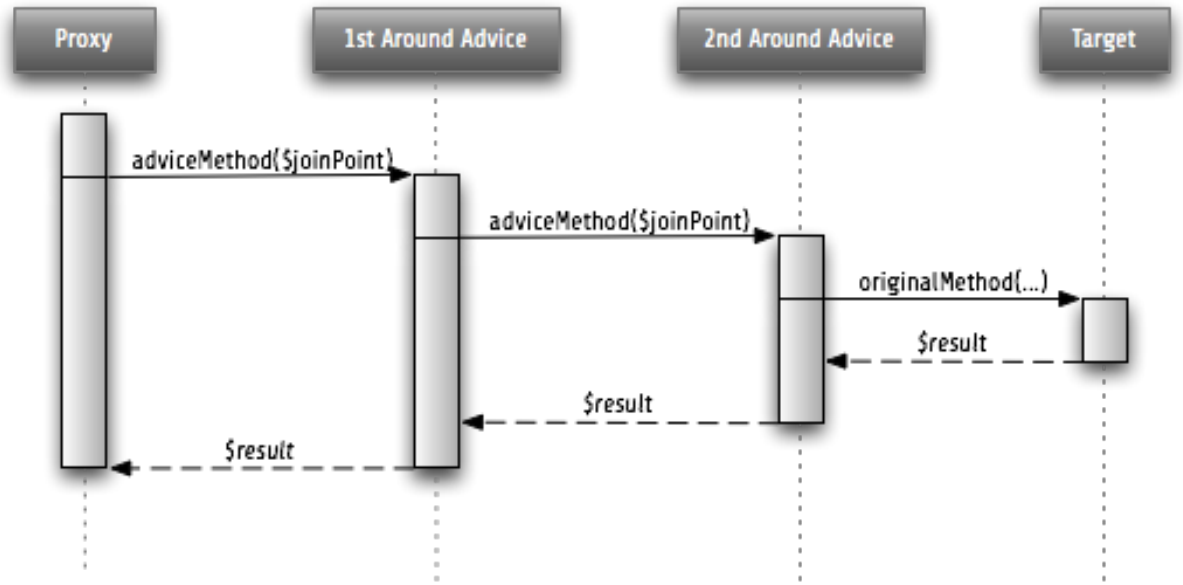

Fig. 29: Control flow of an advice chain

Examples

Let's put our knowledge into practice and start with a simple example. First we would like to log each access to methods within a certain package. The following code will just do that:

Example: Simple logging with aspects:

```
namespace Example\MyPackage;

/**
 * A logging aspect
 *
 * @Flow\Aspect
 */
class LoggingAspect {

        /**
         * @var \Psr\Log\LoggerInterface A logger implementation
         */
        protected $logger;

        /**
         * For logging we need a logger, which we will get injected automatically by
         * the Object Manager
         *
```

(continues on next page)

(continued from previous page)

```
         * @param \Neos\Flow\Log\PsrSystemLoggerInterface $logger The System Logger
         * @return void
         */
        public function injectLogger(\Neos\Flow\Log\PsrSystemLoggerInterface $logger)
↪{
                $this->logger = $logger;
        }

        /**
         * Before advice, logs all access to public methods of our package
         *
         * @param  \Neos\Flow\AOP\JoinPointInterface $joinPoint: The current join␣
↪point
         * @return void
         * @Flow\Before("method(public Example\MyPackage\.*->.*())")
         */
        public function logMethodExecution(\Neos\Flow\AOP\JoinPointInterface
↪$joinPoint) {
                $logMessage = 'The method ' . $joinPoint->getMethodName() . ' in␣
↪class ' .
                        $joinPoint->getClassName() . ' has been called.';
                $this->logger->info($logMessage);
        }
}
```

Note that we are using dependency injection for getting the system logger instance to stay independent from any specific logging implementation. We don't have to care about the kind of logger and where it comes from.

Finally an example for the implementation of an around advice: For a guest book, we want to reject the last name "Sarkosh" (because it should be "Skårhøj"), every time it is submitted. Admittedly you probably wouldn't implement this great feature as an aspect, but it's easy enough to demonstrate the idea. For illustration purposes, we don't define the pointcut expression in place but refer to a named pointcut.

Example: Implementation of an around advice:

```
namespace Example\Guestbook;

/**
 * A lastname rejection aspect
 *
 * @Flow\Aspect
 */
class LastNameRejectionAspect {

        /**
         * A pointcut which matches all guestbook submission method invocations
         *
         * @Flow\Pointcut("method(Example\Guestbook\SubmissionHandlingThingy->
↪submit())")
         */
        public function guestbookSubmissionPointcut() {}

        /**
         * Around advice, rejects the last name "Sarkosh"
         *
         * @param  \Neos\Flow\AOP\JoinPointInterface $joinPoint The current join point
         * @return mixed Result of the target method
```

(continues on next page)

(continued from previous page)

```
         * @Flow\Around("Example\Guestbook\LastNameRejectionAspect->
↪guestbookSubmissionPointcut")
         */
        public function rejectLastName(\Neos\Flow\AOP\JoinPointInterface $joinPoint) {
                if ($joinPoint->getMethodArgument('lastName') === 'Sarkosh') {
                        throw new \Exception('Sarkosh is not a valid last name -␣
↪should be Skårhøj!');
                }
                $result = $joinPoint->getAdviceChain()->proceed($joinPoint);
                return $result;
        }
}
```

Please note that if the last name is correct, we proceed with the remaining links in the advice chain. This is very important to assure that the original (target-) method is finally called. And don't forget to return the result of the advice chain . . .

Introductions

Introductions (also known as Inter-type Declarations) allow to subsequently implement an interface or new properties in a given target class. The (usually) newly introduced methods (required by the new interface) can then be implemented by declaring an advice. If no implementation is defined, an empty placeholder method will be generated automatically to satisfy the contract of the introduced interface.

Interface introduction

Like advices, introductions are declared by annotations. But in contrast to advices, the anchor for an introduction declaration is the class declaration of the aspect class. The annotation tag follows this syntax:

`@Flow\Introduce("PointcutExpression", interfaceName="NewInterfaceName")`

Although the PointcutExpression is just a normal pointcut expression, which may also refer to named pointcuts, be aware that only expressions filtering for classes make sense. You cannot use the method() pointcut designator in this context and will typically take the class() designator instead.

The following example introduces a new interface `NewInterface` to the class `OldClass` and also provides an implementation of the method `newMethod`.

Example: Interface introduction:

```
namespace Example\MyPackage;

/**
 * An aspect for demonstrating introductions
 *
 * Introduces Example\MyPackage\NewInterface to the class Example\MyPackage\OldClass:
 *
 * @Flow\Introduce("class(Example\MyPackage\OldClass)", interfaceName="Example\
↪MyPackage\NewInterface")
 * @Flow\Aspect
 */
class IntroductionAspect {

        /**
         * Around advice, implements the new method "newMethod" of the
```

(continues on next page)

(continued from previous page)

```
         * "NewInterface" interface
         *
         * @param  \Neos\Flow\AOP\JoinPointInterface $joinPoint The current join point
         * @return void
         * @Flow\Around("method(Example\MyPackage\OldClass->newMethod())")
         */
        public function newMethodImplementation(\Neos\Flow\AOP\JoinPointInterface
↪$joinPoint) {
                        // We call the advice chain, in case any other advice is␣
↪declared for
                        // this method, but we don't care about the result.
                $someResult = $joinPoint->getAdviceChain()->proceed($joinPoint);

                $a = $joinPoint->getMethodArgument('a');
                $b = $joinPoint->getMethodArgument('b');
                return $a + $b;
        }
}
```

Trait introduction

Like the interface introductions, also trait introductions are declared by annotation. It even uses the same annotation with a different argument:

`@Flow\Introduce("PointcutExpression", traitName="NewTraitName")`

Again only pointcuts filtering for classes make sense. The traitName must be a fully qualified "class" (trait) name without leading backslash.

The following example introduces a trait `SomeTrait` to the class `MyClass`.

Example: Trait introduction:

```
namespace Example\MyPackage;

/**
 * An aspect for demonstrating trait introduction
 *
 * Introduces Example\MyPackage\SomeTrait to the class Example\MyPackage\MyClass:
 *
 * @Flow\Introduce("class(Example\MyPackage\MyClass)", traitName="Example\MyPackage\
↪SomeTrait")
 * @Flow\Aspect
 */
class TraitIntroductionAspect {
}
```

Property introduction

The declaration of a property introduction anchors to a property inside an aspect.

Form of the declaration:

```
/**
 * @var type
 * @Flow\Introduce("PointcutExpression")
 */
protected $propertyName;
```

The declared property will be added to the target classes matched by the pointcut.

The following example introduces a new property "subtitle" to the class `Example\Blog\Domain\Model\Post`:

Example: Property introduction:

```
namespace Example\MyPackage;

/**
 * An aspect for demonstrating property introductions
 *
 * @Flow\Aspect
 */
class PropertyIntroductionAspect {

        /**
         * @var string
         * @Column(length=40)
         * @Flow\Introduce("class(Example\Blog\Domain\Model\Post)")
         */
        protected $subtitle;

}
```

Implementation details

AOP proxy mechanism

The following diagram illustrates the building process of a proxy class:

2.3.18 Security

Security Framework

All tasks related to security of a Flow application are handled centrally by the security framework. Besides other functionality, this includes especially features like authentication, authorization, channel security and a powerful policy component. This chapter describes how you can use Flow's security features and how they work internally.

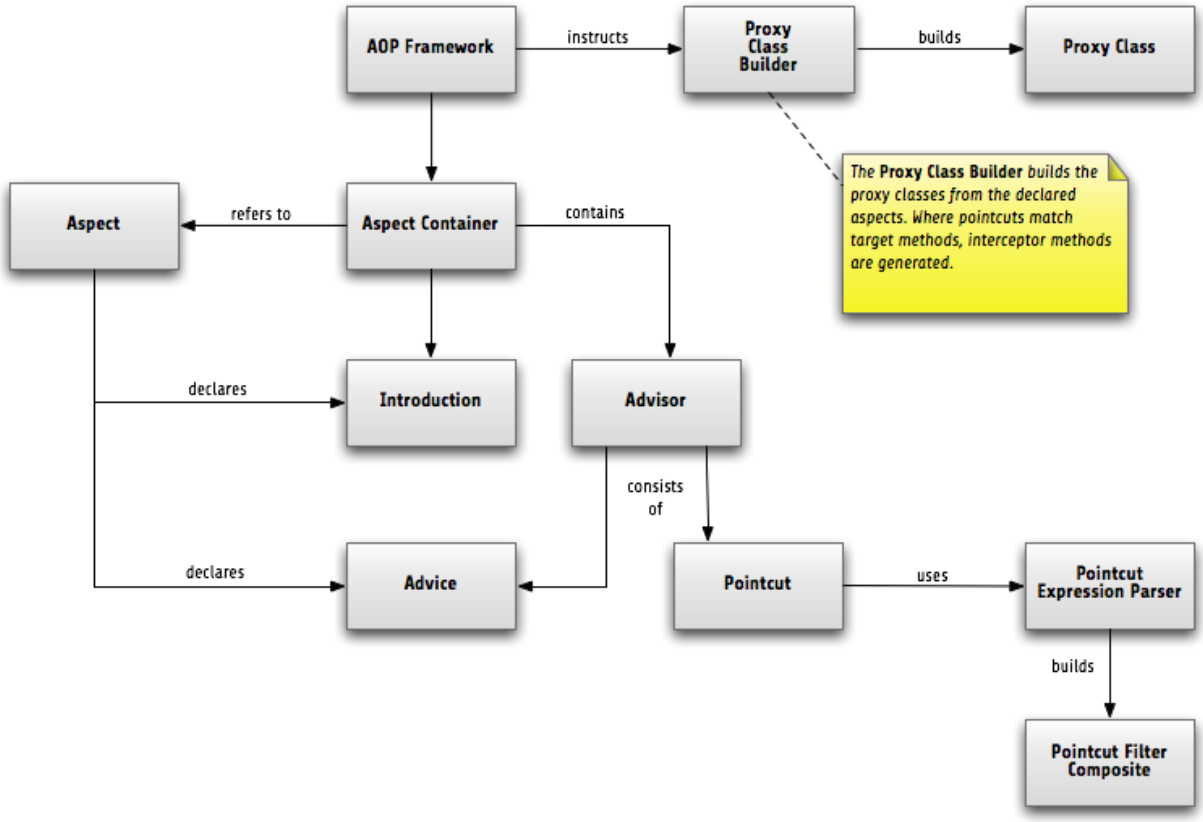

Fig. 30: Proxy building process

Security context

The SECURITY CONTEXT (\Neos\Flow\Security\Context) is initialized as soon as an HTTP request is being dispatched. It lies in session scope and holds context data like the current authentication status. That means, if you need data related to security, the security context (you can get it easily with dependency injection) will be your main information source. The details of the context's data will be described in the next chapters.

Authentication

One of the main things people associate with security is authentication. That means to identify your communication partner - the one sending a request to Flow. Therefore the framework provides an infrastructure to easily use different mechanisms for such a plausibility proof. The most important achievement of the provided infrastructure is its flexible extensibility. You can easily write your own authentication mechanisms and configure the framework to use them without touching the framework code itself. The details are explained in the section *Implementing your own authentication mechanism*.

Using the authentication controller

First, let's see how you can use Flow's authentication features. There is a base controller in the security package: the ABSTRACTAUTHENTICATIONCONTROLLER (\Neos\Flow\Security\Authentication\Controller\AbstractAuthenticationController), which already contains almost everything you need to authenticate an account. This controller has three actions, namely `loginAction()`, `authenticateAction()` and `logoutAction()`. To use authentication in your project you have to inherit from this controller, provide a template for the login action (e.g. a login form) and implement at least the abstract method `onAuthenticationSuccess()`. This method is called if authentication succeeded and will be passed the intercepted request, which triggered authentication. This can be used to resume the original request in order to send the user to the protected area he had tried to access. You may also want to override `onAuthenticationFailure()` to react on login problems appropriately.

Example: Simple authentication controller

```
<?php
namespace Acme\YourPackage\Controller;

use Neos\Flow\Annotations as Flow;
use Neos\Flow\Mvc\ActionRequest;
use Neos\Flow\Security\Authentication\Controller\AbstractAuthenticationController;

class AuthenticationController extends AbstractAuthenticationController {

        /**
         * Displays a login form
         *
         * @return void
         */
        public function indexAction() {
        }

        /**
         * Will be triggered upon successful authentication
         *
         * @param ActionRequest $originalRequest The request that was intercepted by␣
↪the security framework, NULL if there was none
         * @return string
```

(continues on next page)

(continued from previous page)

```
         */
        protected function onAuthenticationSuccess(ActionRequest $originalRequest =
↪NULL) {
                if ($originalRequest !== NULL) {
                        $this->redirectToRequest($originalRequest);
                }
                $this->redirect('someDefaultActionAfterLogin');
        }

        /**
         * Logs all active tokens out and redirects the user to the login form
         *
         * @return void
         */
        public function logoutAction() {
                parent::logoutAction();
                $this->addFlashMessage('Logout successful');
                $this->redirect('index');
        }
}
```

The mechanism that is eventually used to authenticate is implemented in a so called authentication provider. The most common provider (`PersistedUsernamePasswordProvider`) authenticates a user account by checking a username and password against accounts stored in the database.[1]

Example: Configuration of a username/password authentication mechanism in Settings.yaml

```
Neos:
  Flow:
    security:
      authentication:
        providers:
          'SomeAuthenticationProvider':
            provider: 'PersistedUsernamePasswordProvider'
```

This registers the PERSISTEDUSERNAMEPASSWORDPROVIDER (\Neos\Flow\Security\Authentication\Provider\PersistedUsernamePass authentication provider under the name "`SomeAuthenticationProvider`" as the only, global authentication mechanism. To successfully authenticate an account with this provider, you'll obviously have to provide a username and password. This is done by sending two POST variables to the authentication controller. Given there is a route that resolves "your/app/authenticate" to the `authenticateAction()` of the custom `AuthenticationController`, users can be authenticated with a simple login form like the following:

Example: A simple login form

```
<form action="your/app/authenticate" method="post">
   <input type="text"
      name="__
↪authentication[Neos][Flow][Security][Authentication][Token][UsernamePassword][username]
↪" />
   <input type="password"        name="__
↪authentication[Neos][Flow][Security][Authentication][Token][UsernamePassword][password]
↪" />
   <input type="submit" value="Login" />
</form>
```

[1] The details about the `PersistedUsernamePasswordProvider` provider are explained below, in the section about *Authentication mechanisms shipped with Flow*.

After submitting the form the internal authentication process will be triggered and if the provided credentials are valid an account will be authenticated afterwards.[2]

The internal workings of the authentication process

Now that you know, how you can authenticate, let's have a look at the internal process. The following sequence diagram shows the participating components and their interaction:

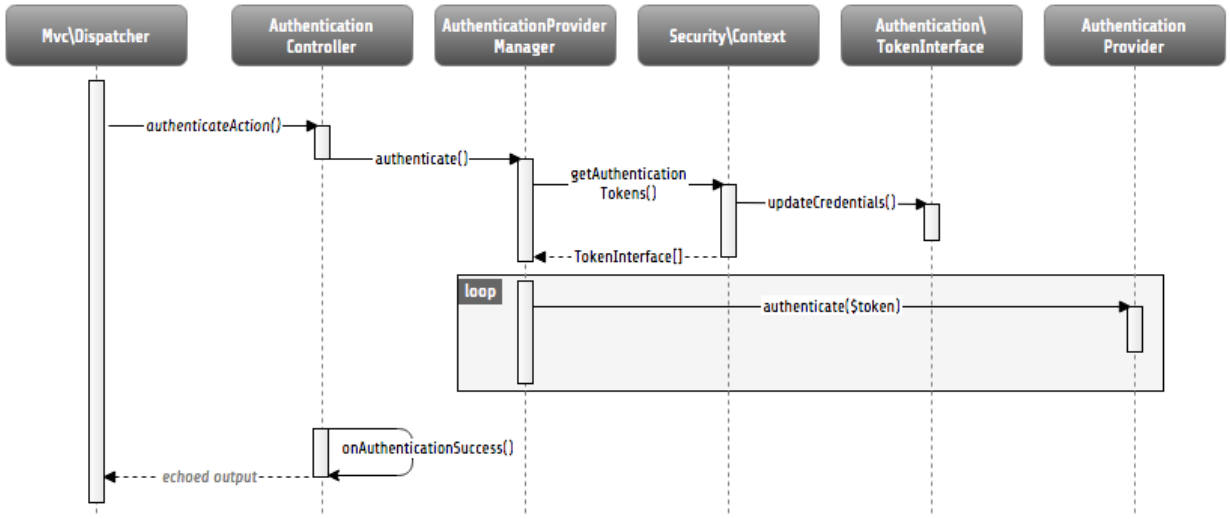

Fig. 31: Internal authentication process

As already explained, the security framework is initialized in the `Neos\Flow\Mvc\Dispatcher`. It intercepts the request dispatching before any controller is called. Regarding authentication, you can see, that a so called authentication token will be stored in the security context and some credentials will be updated in it.

Authentication tokens

An authentication token holds the status of a specific authentication mechanism, for example it receives the credentials (e.g. a username and password) needed for authentication and stores one of the following authentication states in the session.[3]

These constants are defined in the authentication token interface (`Neos\Flow\Security\Authentication\TokenInterface`) and the status can be obtained from the `getAuthenticationStatus()` method of any token.

Tip: If you only want to know, if authentication was successful, you can call the convenience method `isAuthenticated()`.

`NO_CREDENTIALS_GIVEN` This is the default state. The token is not authenticated and holds no credentials, that could be used for authentication.

`WRONG_CREDENTIALS` It was tried to authenticate the token, but the credentials were wrong.

[2] If you don't know any credentials, you'll have to read the section about *Account management*
[3] Well, it holds them in member variables, but lies itself in the security context, which is a class configured as scope session.

AUTHENTICATION_SUCCESSFUL The token has been successfully authenticated.

AUTHENTICATION_NEEDED This indicates, that the token received credentials, but has not been authenticated yet.

Now you might ask yourself, how a token receives its credentials. The simple answer is: It's up to the token, to fetch them from somewhere. The `UsernamePassword` token for example checks for a username and password in POST parameters: By default those parameters are `__authentication[Neos][Flow][Security][Authentication][Token][UsernamePassword][username]` and `__authentication[Neos][Flow][Security][Authentication][Token][UsernamePassword][passwor` This can be changed via `tokenOptions`:

```
Neos:
  Flow:
    security:
      authentication:
        providers:
          'SomeAuthenticationProvider':
            provider: 'PersistedUsernamePasswordProvider'
            tokenOptions:
              usernamePostField: 'auth.username'
              passwordPostField: 'auth.password'
```

With that, the `auth[username]` & `auth[password]` parameters would be evaluated instead – The login template has to be adjusted accordingly of course (see *Using the authentication controller*).

The framework only makes sure that `updateCredentials()` is called on every token, then the token has to set possibly available credentials itself, e.g. from available headers or parameters or anything else you can provide credentials with.

Flow ships with the following authentication tokens:

1. `UsernamePassword`: Extracts username & password from a POST parameter. Options: `usernamePostField` and `passwordPostField`
2. `UsernamePasswordHttpBasic`: Extracts username & password from the the `Authorization` header (Basic auth). This token is sessionless (see below)
3. `PasswordToken`: Extracts password from a POST parameter. Options: `passwordPostField`

But it's really easy to create additional tokens:

Custom authentication tokens

Any class that implements the `TokenInterface` can be used as an authentication token. The following example implements a token that extracts the `Bearer` from an `Authentication` header:

Example: BearerToken.php

```
<?php
namespace Acme\YourPackage;

use Neos\Flow\Annotations as Flow;
use Neos\Flow\Mvc\ActionRequest;
use Neos\Flow\Security\Authentication\Token\AbstractToken;
use Neos\Flow\Security\Authentication\Token\SessionlessTokenInterface;

final class BearerToken extends AbstractToken implements SessionlessTokenInterface
{
    public function updateCredentials(ActionRequest $actionRequest)
```

(continues on next page)

(continued from previous page)

```
    {
        $authorizationHeader = $actionRequest->getHttpRequest()->getHeaderLine(
↪'Authorization');
        if (strncmp($authorizationHeader, 'Bearer ', 7) !== 0) {
            $this->credentials['bearer'] = null;
            $this->authenticationStatus = self::NO_CREDENTIALS_GIVEN;
            return;
        }

        $this->credentials['bearer'] = substr($authorizationHeader, 7);
        $this->authenticationStatus = self::AUTHENTICATION_NEEDED;
    }
}
```

In order to make use of this token, the fully qualified class name has to be specified in the `token` configuration of the corresponding provider:

```
Neos:
  Flow:
    security:
      authentication:
        providers:
          'SomeAuthenticationProvider':
            provider: # ...
            token: 'Acme\YourPackage\BearerToken'
```

Tip: Since the Bearer authentication header is expected to be sent with every request, this token also implements the `SessionlessTokenInterface` (see next section).

For the above token a custom Provider is needed, since the built-in providers won't know how to deal with the `Bearer` token. If a custom token represents username and/or password credentials, they can implement the `UsernamePasswordTokenInterface` or `PasswordTokenInterface`. That way they can be used with the existing `PersistedUsernamePasswordProvider` or `FileBasedSimpleKeyProvider`.

The following example implements a custom token that extracts username and token from a HTTP header that can be specified via options:

Example: UsernamePasswordFromHeaderToken.php

```
<?php
namespace Acme\YourPackage;

use Neos\Flow\Mvc\ActionRequest;
use Neos\Flow\Security\Authentication\Token\AbstractToken;
use Neos\Flow\Security\Authentication\Token\UsernamePasswordTokenInterface;

final class UsernamePasswordFromHeaderToken extends AbstractToken implements␣
↪UsernamePasswordTokenInterface
{
    /**
     * @var string
     */
    private $usernameHeaderName;

    /**
```

(continues on next page)

(continued from previous page)

```
     * @var string
     */
    private $passwordHeaderName;

    public function __construct(array $options)
    {
        $this->usernameHeaderName = $options['usernameHeaderName'] ?? 'X-Username';
        $this->passwordHeaderName = $options['passwordHeaderName'] ?? 'X-Password';
    }

    public function updateCredentials(ActionRequest $actionRequest)
    {
        $username = $actionRequest->getHttpRequest()->getHeaderLine($this->
↪usernameHeaderName);
        $password = $actionRequest->getHttpRequest()->getHeaderLine($this->
↪passwordHeaderName);
        if (empty($username) || empty($password)) {
            $this->credentials = ['username' => null, 'password' => null];
            $this->authenticationStatus = self::NO_CREDENTIALS_GIVEN;
            return;
        }

        $this->credentials = ['username' => $username, 'password' => $password];
        $this->authenticationStatus = self::AUTHENTICATION_NEEDED;
    }

    public function getUsername(): string
    {
        return $this->credentials['username'] ?? '';
    }

    public function getPassword(): string
    {
        return $this->credentials['password'] ?? '';
    }
}
```

This would read username & password from `X-Username` and `X-Password` headers by default, but allow the header names to be specified via `tokenOptions`.

```
Neos:
  Flow:
    security:
      authentication:
        providers:
          DefaultProvider:
            provider: PersistedUsernamePasswordProvider
            token: 'Acme\YourPackage\UsernamePasswordFromHeaderToken'
            tokenOptions:
              usernameHeaderName: 'Some-Header'
              passwordHeaderName: 'Some-Other-Header'
```

The `tokenOptions` will be passed to the constructor of the token.

Note: This serves merely as a simple example of the extension mechanism. It's probably not a good idea to specify

credentials via arbitrary HTTP headers!

Sessionless authentication tokens

By default Flow assumes that a token which has been successfully authenticated needs a session in order to keep being authenticated on the next HTTP request. Therefore, whenever a user sends a `UsernamePassword` token for authentication, Flow will implicitly start a session and send a session cookie.

For authentication mechanisms which don't require a session this process can be optimized. Headers for HTTP Basic Authentication or an API key is sent on every request, so there's no need to start a session for keeping the token. Especially when dealing with REST services, it is not desirable to start a session.

Authentication tokens which don't require a session simply need to implement the SESSIONLESSTOKENINTERFACE (\Neos\Flow\Security\Authentication\Token\SessionlessTokenInterface) marker interface. If a token carries this marker, the Authentication Manager will refrain from starting a session during authentication.

Authentication manager and provider

After the tokens have been initialized the original request will be processed by the resolved controller. Usually this is done by your authentication controller inheriting the AbstractAuthenticationController of Flow, which will call the authentication manager to authenticate the tokens. In turn the authentication manager calls all authentication providers in the configured order. A provider implements a specific authentication mechanism and is therefore responsible for a specific token type. E.g. the already mentioned `PersistedUsernamePasswordProvider` provider is able to authenticate the `UsernamePassword` token.

After checking the credentials, it is the responsibility of an authentication provider to set the correct authentication status (see above) and `Roles` in its corresponding token. The role implementation resides in the `Neos\Flow\Security\Policy` namespace. (see the Policy section for details).

Note: Previously roles were entities, so they were stored in the database. This is no longer the case since Flow 3.0. Instead the active roles will be determined from the configured policies. Creating a new role is as easy as adding a line to your `Policy.yaml`. If you do need to add roles during runtime, you can use the `rolesInitialized` Signal of the POLICYSERVICE (\Neos\Flow\Security\Policy\PolicyService).

Account management

In the previous section you have seen, how accounts can be authenticated in Flow. What was concealed so far is, how these accounts are created or what is exactly meant by the word "account". First of all let's define what accounts are in Flow and how they are used for authentication. Following the OASIS CIQ V3.0[4] specification, an account used for authentication is separated from a user or more general a party. The advantage of this separation is the possibility of one user having more than one account. E.g. a user could have an account for the `UsernamePassword` provider and one account connected to an LDAP authentication provider. Another scenario would be to have different accounts for different parts of your Flow application. Read the next section *Advanced authentication configuration* to see how this can be accomplished.

As explained above, the account stores the credentials needed for authentication. Obviously these credentials are provider specific and therefore every account is only valid for a specific authentication provider. This provider to

[4] The specification can be downloaded from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ciq. The implementation of this specification resides in the "Party" package, which is part of the official Neos distribution.

account connection is stored in a property of the account object named `authenticationProviderName`. Appropriate getters and setters are provided. The provider name is configured in the *Settings.yaml* file. If you look back to the default configuration, you'll find the name of the default authentication provider: `DefaultProvider`. Besides that, each account has another property called `credentialsSource`, which points to the place or describes the credentials needed for this account. This could be an LDAP query string, or in case of the `PersistedUsernamePasswordProvider`, the username, password hash and salt are stored directly in this member variable.

It is the responsibility of the authentication provider to check the given credentials from the authentication token, find the correct account for them[5] and to decide about the authentication status of this token.

Note: In case of a directory service, the real authentication will probably not take place in the provider itself, but the provider will pass the result of the directory service on to the authentication token.

Note: The `DefaultProvider` authentication provider used in the examples is not shipped with Flow, you have to configure all available authentication providers in your application.

Creating accounts

Creating an account is as easy as creating a new account object and add it to the account repository. Look at the following example, which uses the `Neos\Flow\Security\AccountFactory` to create a simple username/password account for the DefaultProvider:

Example: Add a new username/password account

```
$identifier = 'andi';
$password = 'secret';
$roles = array('Acme.MyPackage:Administrator');
$authenticationProviderName = 'DefaultProvider';

$account = $this->accountFactory->createAccountWithPassword($identifier, $password,
↪$roles, $authenticationProviderName);
$this->accountRepository->add($account);
```

The way the credentials are stored internally is completely up to the authentication provider. The `PersistedUsernamePasswordProvider` uses the `Neos\Flow\Security\Cryptography\HashService` to verify the given password. In the example above, the given plaintext password will be securely hashed by the `HashService`. The hashing is the main magic happening in the `AccountFactory` and the reason why we don't create the account object directly. If you want to learn more about secure password hashing in Flow, you should read the section about *Cryptography* below. You can also see, that there is an array of roles added to the account. This is used by the policy system and will be explained in the according section below.

Note: This example expects the account factory and account repository to be available in `$this->accountFactory` and `$this->accountRepository` respectively. If you use this snippet in a command controller, these can be injected very easily by dependency injection.

---

[5] The `AccountRepository` provides a convenient find method called `findActiveByAccountIdentifierAndAuthenticationProviderName()` for this task.

Advanced authentication configuration

Parallel authentication

Now that you have seen all components, taking part in the authentication process, it is time to have a look at some advanced configuration possibilities. Just to remember, here is again the configuration of an authentication provider:

```
security:
  authentication:
    providers:
      'DefaultProvider':
        provider: 'PersistedUsernamePasswordProvider'
```

If you have a closer look at this configuration, you can see, that the word providers is plural. That means, you have the possibility to configure more than one provider and use them in "parallel".

Note: You will have to make sure, that each provider has a unique name. In the example above the provider name is `DefaultProvider`.

Note: You can also disable an authentication provider by setting the provider value to `false` in the YAML configuration. For instance `DefaultProvider: false`.

Example: Configuration of two authentication providers

```
security:
  authentication:
    providers:
      'MyLDAPProvider':
        provider: 'Neos\MyCoolPackage\Security\Authentication\MyLDAPProvider'
        providerOptions: 'Some LDAP configuration options'
      'DefaultProvider':
        provider: 'PersistedUsernamePasswordProvider'
```

This will advice the authentication manager to first authenticate over the LDAP provider and if that fails it will try to authenticate the default provider. So this configuration can be seen as an authentication fallback chain, of course you can configure as many providers as you like, but keep in mind that the order matters.

Note: As you can see in the example, the LDAP provider is provided with some options. These are specific configuration options for each provider, have a look in the detailed description to know if a specific provider needs more options to be configured and which.

Parallel authentication for the same account

Accounts are bound to an authentication provider and by default the `PersistedUsernamePasswordProvider` will only lookup accounts that belong to the provider (i.e. with an `authenticationProviderName` that is equal to the configured provider name, `SomeAuthenticationProvider` in this example. That lookup name can be changed via the `lookupProviderName` option that allows the provider to lookup accounts for a different configuration. This can be useful in order to re-use the same provider & accounts for multiple authentication types, for example classic form-based and HTTP basic auth:

```
Neos:
  Flow:
    security:
      authentication:
        providers:
          'Acme.SomePackage:Default':
            requestPatterns:
              # ...
            provider: PersistedUsernamePasswordProvider
            token: UsernamePassword
            entryPoint: WebRedirect
            entryPointOptions:
              routeValues:
                '@package': Acme.SomePackage
                '@controller': Authentication
                '@action': login

          'Acme.SomePackage:Default.HttpBasic':
            requestPatterns:
              # ...
            provider: PersistedUsernamePasswordProvider
            providerOptions:
              lookupProviderName: 'Acme.SomePackage:Default'
            token: UsernamePasswordHttpBasic
            entryPoint: HttpBasic
```

Multi-factor authentication strategy

There is another configuration option to realize a multi-factor-authentication. It defaults to `oneToken`. A configurable authentication strategy of `allTokens` forces the authentication manager to always authenticate all configured providers and to make sure that every single provider returned a positive authentication status to one of its tokens. The authentication strategy `atLeastOneToken` will try to authenticate as many tokens as possible but at least one. This is helpful to realize policies with additional security only for some resources (e.g. SSL client certificates for an admin backend).

```
configuration:
  security:
    authentication:
      authenticationStrategy: allTokens
```

Reuse of tokens and providers

There is another configuration option for authentication providers called `token`, which can be specified in the provider settings. By this option you can specify which token should be used for a provider. Remember the token is responsible for the credentials retrieval, i.e. if you want to authenticate let's say via username and password this setting enables to to specify where these credentials come from. So e.g. you could reuse the one username/password provider class and specify, whether authentication credentials are sent in a POST request or set in an HTTP Basic authentication header.

Example: Specifying a specific token type for an authentication provider

```
security:
  authentication:
    providers:
      'DefaultProvider':
        provider: 'PersistedUsernamePasswordProvider'
        token: 'UsernamePasswordHttpBasic'
```

Request Patterns

Now that you know about the possibility of configuring more than one authentication provider another scenario may come to your mind. Just imagine an application with two areas: One user area and one administration area. Both must be protected, so we need some kind of authentication. However for the administration area we want a stronger authentication mechanism than for the user area. Have a look at the following provider configuration:

Example: Using request patterns

```
security:
  authentication:
    providers:
      'LocalNetworkProvider':
        provider: 'FileBasedSimpleKeyProvider'
        providerOptions:
          keyName: 'AdminKey'
          authenticateRoles: ['Acme.SomePackage:Administrator']
        requestPatterns:
          'Acme.SomePackage:AdministrationArea':
            pattern: 'ControllerObjectName'
            patternOptions:
              'controllerObjectNamePattern': 'Acme\SomePackage\AdministrationArea\.*'
          'Acme.SomePackage:LocalNetwork':
            pattern: 'Ip'
            patternOptions:
              'cidrPattern': '192.168.178.0/24'
      'MyLDAPProvider':
        provider: 'Neos\MyCoolPackage\Security\Authentication\MyLDAPProvider'
        providerOptions: 'Some LDAP configuration options'
        requestPatterns:
          'Acme.SomePackage:AdministrationArea':
            pattern: 'ControllerObjectName'
            patternOptions:
              'controllerObjectNamePattern': 'Acme\SomePackage\AdministrationArea\.*'
      DefaultProvider:
        provider: 'PersistedUsernamePasswordProvider'
        requestPatterns:
          'Acme.SomePackage:UserArea':
            pattern: 'ControllerObjectName'
```

(continues on next page)

(continued from previous page)

```
            patternOptions:
              'controllerObjectNamePattern': 'Acme\SomePackage\UserArea\.*'
```

Look at the new configuration option `requestPatterns`. This enables or disables an authentication provider, depending on given patterns. The patterns will look into the data of the current request and tell the authentication system, if they match or not. The patterns in the example above will match, if the controller object name of the current request (the controller to be called) matches on the given regular expression. If a pattern does not match, the corresponding provider will be ignored in the whole authentication process. In the above scenario this means, all controllers responsible for the administration area will use the LDAP authentication provider unless the user is on the internal network, in which case he can use a simple password. The user area controllers will be authenticated by the default username/password provider.

Note: You can use more than one pattern in the configuration. Then the provider will only be active, if all patterns match on the current request.

Tip: There can be patterns that match on different data of the request. Just imagine an IP pattern, that matches on the request IP. You could, e.g. provide different authentication mechanisms for people coming from your internal network, than for requests coming from the outside.

Tip: You can easily implement your own pattern. Just implement the interface `Neos\Flow\Security\RequestPatternInterface` and configure the pattern with its full qualified class name.

Available request patterns

Request Pattern	Match criteria	Configuration options	Description
ControllerObjectName	Matches on the object name of the controller that has been resolved by the MVC dispatcher for the current request	`controllerObjectNamePattern`	A regular expression to match on the object name, for example: `controllerObjectNamePattern: 'My\Package\Controller\Admin\.*`
Uri	Matches on the URI of the current request of the current request	`uriPattern`	A regular expression to match on the URI, for example: `uriPattern: '/admin/.*`
Host	Matches on the host part of the current request	`hostPattern`	A wildcard expression to match on the hostname, for example: `hostPattern: '*.mydomain.com'` or `hostPattern: 'www.mydomain.*'`
Ip	Matches on the user IP address of the current request	`cidrPattern`	A CIDR expression to match on the source IP, for example: `cidrPattern: '192.168.178.0/24'` or `cidrPattern: 'fd9e:21a7:a92c:2323::/96'`

Note: The pattern for `Uri` will have slashes escaped and is amended with `^...$` automatically, so do not include

those in your pattern!

Authentication entry points

One question that has not been answered so far is: what happens if the authentication process fails? In this case the authentication manager will throw an `AuthenticationRequired` exception. It might not be the best idea to let this exception settle its way up to the browser, right? Therefore we introduced a concept called authentication entry points. These entry points catch the mentioned exception and should redirect the user to a place where she can provide proper credentials. This could be a login page for the username/password provider or an HTTP header for HTTP authentication. An entry point can be configured for each authentication provider. Look at the following example, that redirects to a login page (Using the `WebRedirect` entry point).

Example: Redirect an ``AuthenticationRequired`` exception to the login page

```
security:
  authentication:
    providers:
      DefaultProvider:
        provider: PersistedUsernamePasswordProvider
        entryPoint: 'WebRedirect'
        entryPointOptions:
          routeValues:
            '@package': 'Your.Package'
            '@controller': 'Authenticate'
            '@action': 'login'
```

Note: Prior to Flow version 1.2 the option `routeValues` was not supported by the WebRedirect entry point. Instead you could provide the option `uri` containing a relative or absolute URI to redirect to. This is still possible, but we recommend to use `routeValues` in order to make your configuration more independent from the routing configuration.

Note: Of course you can implement your own entry point and configure it by using its full qualified class name. Just make sure to implement the `Neos\Flow\Security\Authentication\EntryPointInterface` interface.

Tip: If a request has been intercepted by an `AuthenticationRequired` exception, this request will be stored in the security context. By this, the authentication process can resume this request afterwards. Have a look at the Flow authentication controller if you want to see this feature in action.

Available authentication entry points

Entry Point	Description	Configuration options

WebRedirect	Triggers an HTTP redirect to a given uri or action.	Expects an associative array with either an entry `uri` (obsolete, see Note above), or an array `routeValues`; for example:
		`uri: login/`
		or
		`routeValues:`
		`  '@package': 'Your.`
		`↪Package'`
		`  '@controller':`
		`↪'Authenticate'`
		`  '@action': 'login'`

HttpBasic	Adds a WWW-Authenticate header to the response, which will trigger the browsers authentication form.	Optionally takes an option realm, which will be displayed in the authentication prompt.

Authentication mechanisms shipped with Flow

This section explains the details of each authentication mechanism shipped with Flow. Mainly the configuration options and usage will be exposed, if you want to know more about the entire authentication process and how the components will work together, please have a look in the previous sections.

Simple username/password authentication

Provider

The implementation of the corresponding authentication provider resides in the class `Neos\Flow\Security\Authentication\Provider\PersistedUsernamePasswordProvider`. It is able to authenticate tokens of the type `Neos\Flow\Security\Authentication\Token\UsernamePassword`. It expects a credentials array in the token which looks like that:

```
array(
  'username' => 'admin',
  'password' => 'plaintextPassword'
);
```

It will try to find an account in the `Neos\Flow\Security\AccountRepository` that has the username value as account identifier and fetch the credentials source.

Tip: You should always use the Flow hash service to generate hashes! This will make sure that you really have secure hashes.

The provider will try to authenticate the token by asking the Flow hash service to verify the hashed password against the given plaintext password from the token. If you want to know more about accounts and how you can create them, look in the corresponding section above.

Token

The username/password token is implemented in the class `Neos\Flow\Security\Authentication\Token\UsernamePassword`. It fetches the credentials from the HTTP POST data, look at the following program

listing for details:

```
$postArguments = $this->environment->getRawPostArguments();
$username = \Neos\Utility\ObjectAccess::getPropertyPath($postArguments,
    '__authentication.Neos.Flow.Security.Authentication.Token.UsernamePassword.
↪username');
$password = \Neos\Utility\ObjectAccess::getPropertyPath($postArguments,
    '__authentication.Neos.Flow.Security.Authentication.Token.UsernamePassword.
↪password');
```

Note: The token expects a plaintext password in the POST data. That does not mean, you have to transfer plaintext passwords, however it is not the responsibility of the authentication layer to encrypt the transfer channel. Look in the section about *Application firewall* for any details.

Implementing your own authentication mechanism

One of the main goals for the authentication architecture was to provide an easily extensible infrastructure. Now that the authentication process has been explained, you'll here find the steps needed to implement your own authentication mechanism:

Authentication token

You'll have to provide an authentication token, that implements the interface `Neos\Flow\Security\Authentication\TokenInterface`:

1. The most interesting method is `updateCredentials()`. There you'll get the current request and you'll have to make sure that credentials sent from the client will be fetched and stored in the token.
2. Implement the remaining methods of the interface. These are mostly getters and setters, have a look in one of the existing tokens (for example `Neos\Flow\Security\Authentication\Token\UsernamePassword`), if you need more information.

Tip: You can inherit from the `AbstractToken` class, which will most likely have a lot of the methods already implemented in a way you need them.

Authentication provider

After that you'll have to implement your own authentication mechanism by providing a class, that implements the interface `Neos\Flow\Security\Authentication\AuthenticationProviderInterface`:

1. In the constructor you will get the name, that has been configured for the provider and an optional options array. Basically you can decide on your own which options you need and how the corresponding yaml configuration will look like.
2. Then there has to be a `canAuthenticate()` method, which gets an authentication token and returns a boolean value whether your provider can authenticate that token or not. Most likely you will call `getAuthenticationProviderName()` on the token and check, if it matches the provider name given to you in your provider's constructor. In addition to this, the method `getTokenClassNames()` has to return an array with all authentication token classes, your provider is able to authenticate.
3. All the magic will happen in the `authenticate()` method, which will get an appropriate authentication token. Basically you could do whatever you want in this method, the only thing you'll have to make sure is to set the correct status (possible values are defined as constants in the token interface and explained above). If authentication succeeds you might also want to set an account in the given token, to add some roles to the

current security context. However, here is the recommended way of what should be done in this method and if you don't have really good reasons, you shouldn't deviate from this procedure.

4. Get the credentials provided by the client from the authentication token (`getCredentials()`)
5. Retrieve the corresponding account object from the account repository, which you should inject into your provider by dependency injection. The repository provides a convenient find method for this task: `findActiveByAccountIdentifierAndAuthenticationProviderName()`.
6. The `credentialsSource` property of the account will hold the credentials you'll need to compare or at least the information, where these credentials lie.
7. Start the authentication process (e.g. compare credentials/call directory service/...).
8. Depending on the authentication result, set the correct status in the authentication token, by `calling setAuthenticationStatus()`.
9. Set the account in the authentication token, if authentication succeeded. This will add the roles of this token to the security context.

Tip: You can inherit from the `AbstractProvider` class, which will most likely have a lot of the methods already implemented in a way you need them.

Authorization

This section covers the authorization features of Flow and how those can be leveraged in order to configure fine grained access rights.

Note: With version 3.0 of Flow the security framework was subject to a major refactoring. In that process the format of the policy configuration was adjusted in order to gain flexibility. Amongst others the term `resource` has been renamed to `privilege` and ACLs are now configured directly with the respective role. All changes are covered by code migrations, so make sure to run the `./flow core:migrate` command when upgrading from a previous version.

Privileges

In a complex web application there are different elements you might want to protect. This could be the permission to execute certain actions or the retrieval of certain data that has been stored in the system. In order to distinguish between the different types the concept of `Privilege Types` has been introduced. Privilege Types are responsible to protect the different parts of an application. Flow provides the two generic types `MethodPrivilege` and `EntityPrivilege`, which will be explained in detail in the sections below.

Defining Privileges (Policies)

This section will introduce the recommended and default way of connecting authentication with authorization. In Flow policies are defined in a declarative way. This is very powerful and gives you the possibility to change the security policy of your application without touching any PHP code. The policy system deals with two major objects, which are explained below: `Roles` and `Privilege Targets`. All policy definitions are configured in the `Policy.yaml` files.

Privilege Targets

In general a Privilege Target is the definition pointing to something you want to protect. It consists of a **Privilege Type**, a **unique name** and a **matcher expression** defining which things should be protected by this target.

The privilege type defines the nature of the element to protect. This could be the execution of a certain action in your system, the retrieval of objects from the database, or any other kind of action you want to supervise in your application. The following example defines a Privilege Target for the `MethodPrivilege` type to protect the execution of some methods.

Example: privilege target definition in the Policy.yaml file

```
privilegeTargets:

  'Neos\Flow\Security\Authorization\Privilege\Method\MethodPrivilege':

    'Acme.MyPackage:RestrictedController.customerAction':
      matcher: 'method(Acme\MyPackage\Controller\RestrictedController->
↪customerAction())'

    'Acme.MyPackage:RestrictedController.adminAction':
      matcher: 'method(Acme\MyPackage\Controller\RestrictedController->adminAction())'

    'Acme.MyPackage:editOwnPost':
      matcher: 'method(Acme\MyPackage\Controller\PostController->editAction(post.
↪owner == current.userService.currentUser))'
```

Privilege targets are defined in the `Policy.yaml` file of your package and are grouped by their respective types, which are define by the fully qualified classname of the privilege type to be used (e.g. `Neos\Flow\Security\Authorization\Privilege\Method\MethodPrivilege`). Besides the type each privilege target is given a unique name[6] and a so called matcher expression, which would be a pointcut expression in case of the Method Privilege.

Looking back to the example above, there are three privilege targets defined, matching different methods, which should be protected. You even can use runtime evaluations to specify method arguments, which have to match when the method is called.

Roles and privileges

In the section about authentication roles have been introduced. Roles are attached to a user's security context by the authentication system, to determine which privileges should be granted to her. I.e. the access rights of a user are decoupled from the user object itself, making it a lot more flexible, if you want to change them. In Flow roles are defined in the `Policy.yaml` files, and are unique within your package namespace. The full identifier for a role would be `<PackageKey>:<RoleName>`.

For the following examples the context is the `Policy.yaml` file of the `Acme.MyPackage` package.

Following is an example of a simple policy configuration, that will proclaim the roles `Acme.MyPackage:Administrator`, `Acme.MyPackage:Customer`, and `Acme.MyPackage:PrivilegedCustomer` to the system and assign certain privileges to them.

[6] By convention the privilege target identifier is to be prefixed with the respective package key to avoid ambiguity.

Example: Simple roles definition in the Policy.yaml file

```
roles:
  'Acme.MyPackage:Administrator':
    privileges: []

  'Acme.MyPackage:Customer':
    privileges: []

  'Acme.MyPackage:PrivilegedCustomer':
    parentRoles: ['Acme.MyPackage:Customer']
    privileges: []
```

The role `Acme.MyPackage:PrivilegedCustomer` is configured as a sub role of `Acme.MyPackage:Customer`, for example it will inherit the privileges from the `Acme.MyPackage:Customer` role.

Flow will always add the magic `Neos.Flow:Everybody` role, which you don't have to configure yourself. This role will also be present, if no account is authenticated.

Likewise, the magic role `Neos.Flow:Anonymous` is added to the security context if no user is authenticated and `Neos.Flow:AuthenticatedUser` if there is an authenticated user.

Defining Privileges and Permissions

The last step is to connect privilege targets with roles by assigning permissions. Let's extends our roles definition accordingly:

Example: Defining privileges and permissions

```
roles:
  'Acme.MyPackage:Administrator':
    privileges:
      -
        privilegeTarget: 'Acme.MyPackage:RestrictedController.customerAction'
        permission: GRANT
      -
        privilegeTarget: 'Acme.MyPackage:RestrictedController.adminAction'
        permission: GRANT
      -
        privilegeTarget: 'Acme.MyPackage:RestrictedController.editOwnPost'
        permission: GRANT

  'Acme.MyPackage:Customer':
    privileges:
      -
        privilegeTarget: 'Acme.MyPackage:RestrictedController.customerAction'
        permission: GRANT

  'Acme.MyPackage:PrivilegedCustomer':
    parentRoles: ['Acme.MyPackage:Customer']
    privileges:
      -
        privilegeTarget: 'Acme.MyPackage:RestrictedController.editOwnPost'
        permission: GRANT
```

This will end up in `Administrators` being able to call all the methods matched by the three privilege targets from above. However, `Customers` are only able to call the `customerAction`, while `PrivilegedCustomers` are also allowed to edit their own posts. And all this without touching one line of PHP code, isn't that convenient?

Privilege evaluation

Privilege evaluation is a really complex task, when you think carefully about it. However, if you remember the following two rules, you will have no problems or unexpected behaviour when writing your policies:

1. If a DENY permission is configured for one of the user's roles, access will be denied no matter how many grant privileges there are in other roles.
2. If no privilege has been defined for any of the user's roles, access will be denied implicitly.

This leads to the following best practice when writing policies: Use the implicit deny feature as much as possible! By defining privilege targets, all matched subjects (methods, entities, etc.) will be denied implicitly. Use GRANT permissions to allow access to them for certain roles. The use of a DENY permission should be the ultimate last resort for edge cases. Be careful, there is no way to override a DENY permission, if you use it anyways!

Using privilege parameters

To explain the usage of privilege parameters, imagine the following scenario: there is an invoice service which requires the approval of invoices with an amount greater than 100 Euros. Depending on the invoice amount different roles are allowed to approve an invoice or not. The respective MethodPrivilege could look like the following:

```
privilegeTargets:

  'Neos\Flow\Security\Authorization\Privilege\Method\MethodPrivilege':

    'Acme.MyPackage:InvoiceService.ApproveInvoiceGreater100Euros':
      matcher: 'method(Acme\MyPackage\Controller\InvoiceService->approve(invoice.
↪amount > 100))'

    'Acme.MyPackage:InvoiceService.ApproveInvoiceGreater1000Euros':
      matcher: 'method(Acme\MyPackage\Controller\InvoiceService->approve(invoice.
↪amount > 1000))'

roles:
  'Acme.MyPackage:Employee':
    privileges:
      -
        privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoiceGreater100Euros'
        permission: GRANT
      -
        privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoiceGreater1000Euros
↪'
        permission: DENY

  'Acme.MyPackage:CEO':
    privileges:
      -
        privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoiceGreater100Euros'
        permission: GRANT
      -
        privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoiceGreater1000Euros
↪'
        permission: GRANT
```

While this example policy is pretty straight forward, you can imagine, that introducing further approval levels will end up in a lot of specific privilege targets to be created. For this we introduced a concept called privilege parameters. The following Policy expresses the exact same functionality as above:

```
privilegeTargets:

  'Neos\Flow\Security\Authorization\Privilege\Method\MethodPrivilege':

    'Acme.MyPackage:InvoiceService.ApproveInvoice':
      matcher: 'method(Acme\MyPackage\Controller\InvoiceService->approve(invoice.
↪amount > {amount}))'
      parameters:
        amount:
          className: 'Neos\Flow\Security\Authorization\Privilege\Parameter\
↪StringPrivilegeParameter'

  roles:
    'Acme.MyPackage:Employee':
      privileges:
        -
          privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoice'
          parameters:
            amount: 100
          permission: GRANT
        -
          privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoice'
          parameters:
            amount: 1000
          permission: DENY

    'Acme.MyPackage:CEO':
      privileges:
        -
          privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoice'
          parameters:
            amount: 100
          permission: GRANT
        -
          privilegeTarget: 'Acme.MyPackage:InvoiceService.ApproveInvoice'
          parameters:
            amount: 1000
          permission: GRANT
```

As you can see we saved one privilege target definition. The specific amount will not be defined in the privilege target anymore, but is passed along as parameter with the permission for a specific role. Of course, a privilege target can have an arbitrary number of parameters, which can be filled by their names within the roles' privilege configuration.

Internal workings of method invocation authorization (MethodPrivilege)

One of the generic privilege types shipped with Flow is the MethodPrivilege, which protects the invocation of certain methods. By controlling, which methods are allowed to be called and which not, it can be globally ensured, that no unprivileged action will be executed at any time. This is what you would usually do, by adding an access check at the beginning of your privileged method. In Flow, there is the opportunity to enforce these checks without touching the actual method at all. Obviously Flow's AOP features are used to realize this completely new perspective on authorization. If you want to learn more about AOP, please refer to the corresponding chapter in this reference.

First, let's have a look at the following sequence diagram to get an overview of what is happening when an authorization decision is formed and enforced:

As already said, the whole authorization starts with an intercepted method, or in other words with a method that should be protected and only be callable by privileged users. In the chapter about AOP you've already read, that every method

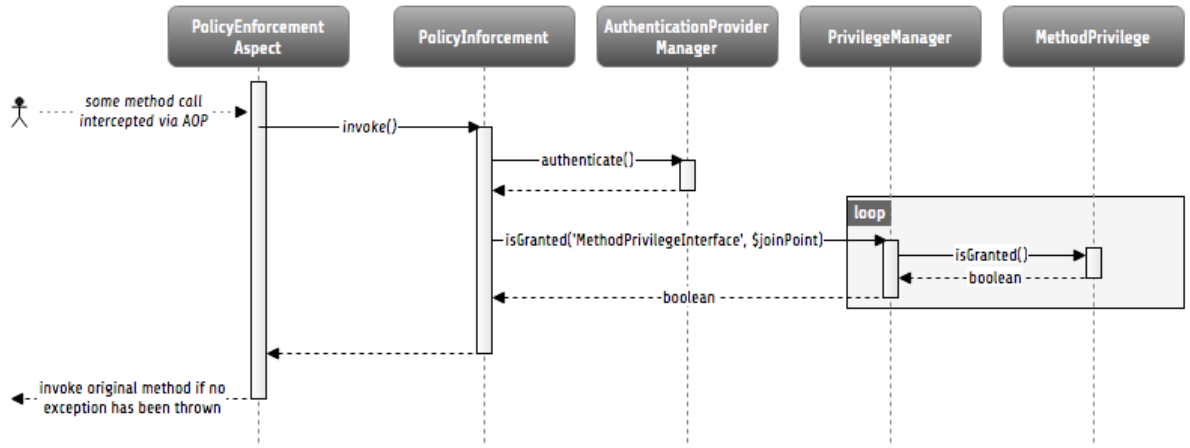

Fig. 32: How an authorization decision is formed and enforced in Flow

interception is implemented in a so called advice, which resides in an aspect class. Here we are: the `Neos\Flow\Security\Aspect\PolicyEnforcementAspect`. Inside this aspect there is the `enforcePolicy()` advice, which hands over to Flow's authorization components.

The next thing to be called is a security interceptor. This interceptor calls the authentication manager before it continues with the authorization process, to make sure that the authentication status is up to date. Then the privilege manager is called, which has to decide, if calling the intercepted method is granted. If not an access denied exception is thrown by the security interceptor.

The privilege manager simply checks all MethodPrivileges matching the respective method invocation and evaluates the permissions according to the privilege evaluation strategy explained in the previous section.

Content security (EntityPrivilege)

To restrict the retrieval of Doctrine entities stored in the database, Flow ships the generic EntityPrivilege. This privilege type enables you to hide certain entities from certain users. By rewriting the queries issued by the Doctrine ORM, persisted entities a users is not granted to read, are simply not returned from the database. For the respective user it looks like these entities are not existing at all.

The following example shows the matcher syntax used for entity privilege targets:

```
'Neos\Flow\Security\Authorization\Privilege\Entity\Doctrine\EntityPrivilege':

  'Acme.MyPackage.RestrictableEntity.AllEntitiesOfTypeRestrictableEntity':
    matcher: 'isType("Acme\MyPackage\RestrictableEntity")'

  'Acme.MyPackage.HiddenEntities':
    matcher: 'isType("Acme\MyPackage\RestrictableEntity") && TRUE == property("hidden
↪")'

  'Acme.MyPackage.OthersEntities':
    matcher: 'isType("Acme\MyPackage\RestrictableEntity") && !(property("ownerAccount
↪").equals("context.securityContext.account")) && property("ownerAccount") != NULL'
```

EEL expressions are used to target the respective entities. You have to define the entity type, can match on property values and use global objects for comparison.

Global objects (by default the current `SecurityContext` imported as `securityContext`) are registered in the *Settings.yaml* file in `aop.globalObjects`. That way you can add your own as well.

You also can walk over entity associations to compare properties of related entities. The following examples, taken from the functional tests, show some more advanced matcher statements:

```
'Neos\Flow\Security\Authorization\Privilege\Entity\Doctrine\EntityPrivilege':

  'Acme.MyPackage.RelatedStringProperty':
    matcher: 'isType("Acme\MyPackage\EntityA") && property("relatedEntityB.stringValue
↪") == "Admin"'

  'Acme.MyPackage.RelatedPropertyComparedWithGlobalObject':
   matcher: 'isType("Acme\MyPackage\EntityA") && property("relatedEntityB.ownerAccount
↪") != "context.securityContext.account" && property("relatedEntityB.ownerAccount") !
↪= NULL'

  'Acme.MyPackage.CompareStringPropertyWithCollection':
    matcher: 'isType("Acme\MyPackage\EntityC") && property("simpleStringProperty").
↪in(["Andi", "Robert", "Karsten"])'

  'Acme.MyPackage.ComparingWithObjectCollectionFromGlobalObjects':
    matcher: 'isType("Acme\MyPackage\EntityC") && property("relatedEntityD").in(
↪"context.someGloablObject.someEntityDCollection")'
```

Warning: When using class inheritance for your entities, entity privileges will only work with the root entity type. For example, if your entity `Acme\MyPackage\EntityB` extends `Acme\MyPackage\EntityA`, the expression `isType("Acme\MyPackage\EntityB")` will never match. This is a limitation of the underlying Doctrine filter API.

Warning: Custom Global Objects should implement CacheAwareInterface

If you have custom global objects (as exposed through *Neos.Flow.aop.globalObjects*) which depend on the current user (security context), ensure they implement *CacheAwareInterface* and change depending on the relevant access restrictions you want to provide.

The cache identifier for the global object will be included in the Security Context Hash, ensuring that the Doctrine query cache and all other places caching with security in mind will correctly create separate cache entries for the different access restrictions you want to create.

As an example, if your user has a "company" assigned, and depending on the company, your should only see your "own" records, you need to: Implement a custom context object, register it in *Neos.Flow.aop.globalObjects* and make it implement *CacheAwareInterface*:

```
/**
 * @Flow\Scope("singleton")
 */
class UserInformationContext implements CacheAwareInterface
{
    /**
     * @Flow\Inject
     * @var Context
     */
    protected $securityContext;

    /**
```

```
     * @Flow\Inject
     * @var PersistenceManagerInterface
     */
    protected $persistenceManager;

    /**
     * @return Company
     */
    public function getCompany() {
        $account = $this->securityContext->getAccount();
        $company = // find your $company depending on the account;
        return $company;
    }

    /**
     * @return string
     */
    public function getCacheEntryIdentifier()
    {
        $company = $this->getCompany();

        return $this->persistenceManager->getIdentifierByObject($company);
    }
```

Internal workings of entity restrictions (EntityPrivilege)

Internally the Doctrine filter API is used to add additional SQL constraints to all queries issued by the ORM against the database. This also ensures to rewrite queries done while lazy loading objects, or DQL statements. The responsible filter class `Neos\Flow\Security\Authorization\Privilege\Entity\Doctrine\SqlFilter` uses various `ConditionGenerators` to create the needed SQL. It is registered als Doctrine filter with the name `Flow_Security_Entity_Filter` in Flow's Settings.yaml file.

The evaluation of entity restrictions is analog to the MethodPrivilege from above. This means entities matched by a privilege target are implicitly denied and are therefore hidden from the user. By adding a grant permission for a privilege target, this role will be able to retrieve the respective objects from the database. A DENY permission will override any GRANT permission, nothing new here. Internally we add SQL where conditions excluding matching entities for all privilege targets that are not granted to the current user.

Warning: Custom SqlFilter implementations - watch out for data privacy issues!

If using custom SqlFilters, you have to be aware that the SQL filter is cached by doctrine, thus your SqlFilter might not be called as often as you might expect. This may lead to displaying data which is not normally visible to the user!

Basically you are not allowed to call *setParameter* inside *addFilterConstraint*; but setParameter must be called *before* the SQL query is actually executed. Currently, there's no standard Doctrine way to provide this; so you manually can receive the filter instance from *$entityManager->getFilters()->getEnabledFilters()* and call *setParameter()* then.

Alternatively, you can use the mechanism from above, where you register a global context object in *Neos.Flow.aop.globalObjects* and use it to provide additional identifiers for the caching; effectively seggregating the Doctrine cache some more.

Creating your custom privilege

Creating your own privilege type usually has one of the two purposes: # You want to define the existing privileges with your own domain specific language (DSL). # There is a completely new privilege target (neither method calls, nor persisted entities) that needs to be protected.

The first use case can be implemented by inheriting from one of the existing privilege classes. The first step to change the expression syntax is to override the method `matchesSubject(...)`. This method gets a privilege subject object (e.g. a JoinPoint for method invocations) and decides whether this privilege (defined by the matcher expression) matches this subject by returning a boolean. In this method you can therefore implement your custom matching logic, working with your very own domain specific matcher syntax. Of course the existing EEL parser can be used to realize DSLs, but in the end thats totally up to you what to use here.

Tip: To use privilege parameters (see section above), you can use `getParsedMatcher()` from the `AbstractPrivilege`.

The second step is dependant on the privilege type you are extending. This is the implementation of the actual enforcement of the permissions defined by this type.

In case of the MethodPrivilege, you'll also have to override `getPointcutFilterComposite()` to provide the AOP framework with the needed information about which methods have to be intercepted during compile time.

In case of the EntityPrivilege permissions are not enforced directly with the entities, but by changing SQL queries. One could says the database is responsible to enforce the rules by evaluating the SQL. The additional SQL is returned by the EnitiyPrivilege's method `getSqlConstraint()`, which of course can be overriden to support an alternative matcher syntax.

Tip: You might still want to use the existing SQL generators, as this is where the hard lowlevel magic is happening. You can compose your constraint logic by these generator objects in a nice programmatical way.

Coming back to the second use case to create your completely custom privilege type, you also have to implement a privilege class with the two functionalities from above:

1. Create your custom privilege subject as a wrapper object for whatever things you want to protect. Corresponding to this object you'll have to implement the `matchesSubject(...)` method of your custom privilege class.
2. Additionally the permissions have to be enforced. This is totally up to your privilege type, or in other words your use case. Feel free to add custom methods to your privilege class to help you enforcing the new privilege (equivalent to generation of SQL or pointcut filters in the entity or method privilege type, respectively).

Retrieving permission and status information

Besides enforcing the policy it is also important to find out about permissions beforehand, to be able to react on not permitted actions before permissions are actually enforced. To find out about permissions, the central privilege manager (`Neos\Flow\Security\Authorization\PrivilegeManager`) can be asked for different things:

1. If the user with the currently authenticated roles is granted for a given subject: `isGranted(...)`. The subject depends on the privilege type, which bring their specific privilege subject implementations. In case of the MethodPrivilege this would be the concrete method invocation (`JoinPoint`).
2. If the user with the currently authenticated roles is granted for a given privilege target (no matter which privilege type it is): `isPrivilegeTargetGranted(...)`

3. The privilege manager also provides methods to calculate the result for both types of information with different roles. By this one can check what would happen if the user had different roles than currently authenticated: `isGrantedForRoles(...)` and `isPrivilegeTargetGrantedForRoles(...)`

Fluid (view) integration

As already stated it is desirable to reflect the policy rules in the view, e.g. a button or link to delete a customer should not be shown, if the user has not the privilege to do so. If you are using the recommended Fluid templating engine, you can simply use the security view helpers shipped with Fluid. Otherwise you would have to ask the privilege manager - as stated above - for the current privilege situation and implement the view logic on your own. Below you'll find a short description of the available Fluid view helpers.

`ifAccess` view helper

This view helper implements an ifAccess/else condition, have a look at the following example, which should be more or less self-explanatory:

Example: the ifAccess view helper

```
<f:security.ifAccess privilegeTarget="somePrivilegeTargetIdentifier">
   This is being shown in case you have access to the given privilege target
</f:security.ifAccess>

<f:security.ifAccess privilegeTarget="somePrivilegeTargetIdentifier">
   <f:then>
      This is being shown in case you have access.
   </f:then>
   <f:else>
      This is being displayed in case you do not have access.
   </f:else>
</f:security.ifAccess>
```

As you can imagine, the main advantage is, that the view will automatically reflect the configured policy rules, without the need of changing any template code.

`ifHasRole` view helper

This view helper is pretty similar to the `ifAccess` view helper, however it does not check the access privilege for a given privilege target, but the availability of a certain role. For example you could check, if the current user has the `Administrator` role assigned:

Example: the ifHasRole view helper

```
<f:security.ifHasRole role="Administrator">
   This is being shown in case you have the Administrator role (aka role).
</f:security.ifHasRole>

<f:security.ifHasRole role="Administrator">
   <f:then>
      This is being shown in case you have the role.
   </f:then>
   <f:else>
      This is being displayed in case you do not have the role.
```

(continues on next page)

(continued from previous page)

```
   </f:else>
</f:security.ifHasRole>
```

The `ifHasRole` view helper will automatically add the package key from the current controller context. This means that the examples above will only render the 'then part' if the user has the `Administrator` role of the package your template belongs to. If you want to check for a role from a different package you can use the full role identifier or specify the package key with the `packageKey` attribute:

Example: check for a role from a different package

```
<f:security.ifHasRole role="Acme.SomeOtherPackage:Administrator">
   This is being shown in case you have the Administrator role (aka role).
</f:security.ifHasRole>

<f:security.ifHasRole role="Administrator" packageKey="Acme.SomeOtherPackage">
   This is being shown in case you have the Administrator role (aka role).
</f:security.ifHasRole>
```

`ifAuthenticated` view helper

There are cases where it doesn't matter which permissions or roles a user has, it is simply needed to differentiate between authenticated users and anonymous users in general. In these cases the `ifAuthenticated` view helper will be the method of choice:

Example: check if a user is authenticated

```
<f:security.ifAuthenticated>
  <f:then>
    This is being shown in case a user is authenticated
  </f:then>
  <f:else>
    This is being displayed in case no user is authenticated
  </f:else>
</f:security.ifAuthenticated>
</code>
```

Commands to analyze the policy

Flow ships different commands to analyze the configured policy:

1. security:showunprotectedactions: This command lists all controller actions not covered by any privilege target in the system. It helps to find out which actions will be publicly available without any security interception in place.
2. security:showmethodsforprivilegetarget: To test matchers for method privilege, this command lists all methods covered by a given privilege target. Of course this command can only be used with privilege targets of type MethodPrivilege.
3. security:showeffectivepolicy: This command lists the effective permissions for all available privilege targets of the given type (entity or method) in the system. To evaluate these permission the respective roles have to be passed to the command.

Application firewall

Besides the privilege powered authorization, there is another line of defense: the filter firewall. This firewall is triggered directly when a request arrives in the MVC dispatcher. The request is analyzed and can be blocked/filtered out. This adds a second level of security right at the beginning of the whole framework run, which means that a minimal amount of potentially insecure code will be executed before that.

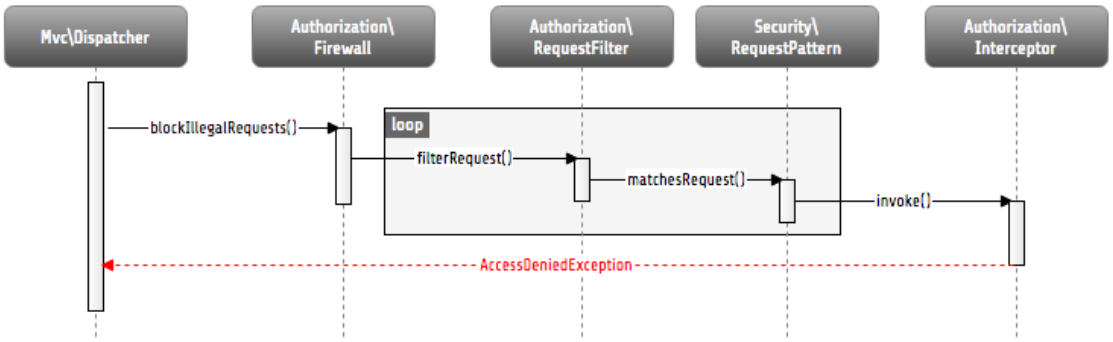

Fig. 33: Blocking request with Flow's filter firewall

Blocking requests with the firewall is not a big thing at all, basically a request filter object is called, which consists of a request pattern and a security interceptor. The simple rule is: if the pattern matches on the request, the interceptor is invoked. *Request Patterns* are also used by the authentication components and are explained in detail there. Talking about security interceptors: you already know the policy enforcement interceptor, which triggers the authorization process. Here is a table of available interceptors, shipped with Flow:

Note: Of course you can implement your own interceptor. Just make sure to implement the interface: `Neos\Flow\Security\Authorization\InterceptorInterface`.

Flow's built-in security interceptors

Security interceptor	Invocation action
PolicyEnforcement	Triggers the authorization process as described one section above.
RequireAuthentication	Calls the authentication manager to authenticate all active tokens for the current request.

Of course, you are able to configure as many request filters as you like. Have a look at the following example to get an idea how a firewall configuration will look like:

Example: Firewall configuration in the Settings.yaml file

```
Neos:
  Flow:
    security:
      firewall:
        rejectAll: FALSE

        filters:
          'Some.Package:AllowedUris':
            pattern:  'Uri'
            patternOptions:
              'uriPattern': '/some/url/.*'
```

(continues on next page)

(continued from previous page)

```
      interceptor:  'AccessGrant'
    'Some.Package:BlockedUris':
      pattern:  'Uri'
      patternOptions:
        'uriPattern': '/some/url/blocked.*'
      interceptor:  'AccessDeny'
    'Some.Package:BlockedHosts':
      pattern:  'Host'
      patternOptions:
        'hostPattern': 'static.mydomain.*'
      interceptor:  'AccessDeny'
    'Some.Package:AllowedIps':
      pattern:  'Ip'
      patternOptions:
        'cidrPattern': '192.168.178.0/24'
      interceptor:  'AccessGrant'
    'Some.Package:CustomPattern':
      pattern:  'Acme\MyPackage\Security\MyOwnRequestPattern'
      patternOptions:
        'someOption': 'some value'
        'someOtherOption': 'some other value'
      interceptor:  'Acme\MyPackage\Security\MyOwnSecurityInterceptor'
```

As you can see, you can easily use your own implementations for request patterns and security interceptors.

Note: You might have noticed the `rejectAll` option. If this is set to `true`, only request which are explicitly allowed by a request filter will be able to pass the firewall.

CSRF protection

A special use case for the filter firewall is CSRF protection. A custom CSRF filter is installed and active by default. It checks every non-safe request (requests are considered safe, if they do not manipulate any persistent data) for a CSRF token and blocks the request if the token is invalid or missing.

Note: Besides safe requests CSRF protection is also skipped for requests with an anonymous authentication status, as these requests are considered publicly callable anyways.

The needed token is automatically added to all URIs generated in Fluid forms, sending data via POST, if any account is authenticated. To add CSRF tokens to URIs, e.g. used for AJAX calls, Fluid provides a special view helper, called `Security.CsrfTokenViewHelper`, which makes the currently valid token available for custom use in templates. In general, you can retrieve the token by calling `getCsrfProtectionToken` on the security context.

Tip: There might be actions, which are considered non-safe by the framework but still cannot be protected by a CSRF token (e.g. authentication requests, send via HTTP POST). For these special cases you can tag the respective action with the `@Flow\SkipCsrfProtection` annotation. Make sure you know what you are doing when using this annotation, it might decrease security for your application when used in the wrong place!

Channel security

Currently, channel security is not a specific feature of Flow. Instead, you have to make sure to transfer sensitive data, like passwords, over a secure channel. This is e.g. to use an SSL connection.

Cryptography

Hash service

Creating cryptographically secure hashes is a crucial part to many security related tasks. To make sure the hashes are built correctly Flow provides a central hash service `Neos\Flow\Security\Cryptography\HashService`, which brings well tested hashing algorithms to the developer. We highly recommend using this service to make sure hashes are securely created.

Flow's hash services provides you with functions to generate and validate HMAC hashes for given strings, as well as methods for hashing passwords with different hashing strategies.

RSA wallet service

Flow provides a so called RSA wallet service to manage public/private key encryption. The idea behind this service is to store private keys securely within the application by only exposing the public key via API. The default implementation shipped with Flow is based on the OpenSSL functions shipped with PHP: `Neos\Flow\Security\Cryptography\RsaWalletServicePhp`.

The service can either create new key pairs itself, while returning the fingerprint as identifier for this key pair. This identifier can be used to export the public key, decrypt and encrypt data or sign data and verify signatures.

To use existing keys the following commands can be used to import keys to be stored and used within the wallet:

- security:importpublickey
- security:importprivatekey

2.3.19 Internationalization & Localization Framework

Internationalization (also known as i18n) is the process of designing software so that it can be easily (i.e. without any source code modifications) adapted to various languages and regions. Localization (also known as L10n) is the process of adapting internationalized software for a specific language or region (e.g. by translating text, formatting date or time).

Basics

Locale class

Instances of `\Neos\Flow\I18n\Locale` class are fundamental for the whole i18n and L10n functionality. They are used to specify what language should be used for translation, how date and time should be formatted, and so on. They can be treated as simple wrappers for locale identifiers (like *de* or *pl_PL*). Many methods from the i18n framework accept Locale objects as a optional parameter - if not provided, the default `Locale` instance for a Flow installation will be used.

You can create a `Locale` object for any valid locale identifier (specified by RFC 4646), even if it is not explicitly meant to be supported by the current Flow installation (i.e. there are no localized resources for this locale). This can be useful, because Flow uses the *Common Locale Data Repository* (CLDR), so each Flow installation knows how to localize numbers, date, time and so on to almost any language and region on the world.

Additionally Flow creates a special collection of available `Locale` objects. Those can either be configured explicitly in settings via `Neos.Flow.i18n.availableLocales` or they are automatically generated by scanning the filesystem for any localized resources. You can use the i18n service API to obtain these verified `Locale` objects.

Note: You can configure which folders Flow should scan for finding available locales through the `Neos.Flow.i18n.scan.includePaths` setting. This is useful to restrict the scanning to specific paths when you have a big file structure in your package `Resources`. You can also exclude folders through `Neos.Flow.i18n.scan.excludePatterns`. By default the `Public` and `Private/Translations` folders, except 'node_modules', 'bower_components' and any folder starting with a dot will be scanned.

Locales are organized in a hierarchy. For example, *en* is a parent of *en_US* which is a parent of *en_US_POSIX*. Thanks to the hierarchical relation resources can be automatically shared between related resources. For example, when you request a *foobar* item for *en_US* locale, and it does not exist, but the item does exist for the *en* locale, it will be used.

Common Locale Data Repository

Flow comes bundled with the CLDR (*Common Locale Data Repository*). It's an Unicode project with the aim to provide a systematic representation of data used for the localization process (like formatting numbers or date and time). The i18n framework provides a convenient API to access this data.

Note: For now Flow covers only a subset of the CLDR data. For example, only the Gregorian calendar is supported for date and time formatting or parsing.

Detecting user locale

The `Detector` class can be used for matching one of the available locales with locales accepted by the user. For example, you can provide the *AcceptLanguage* HTTP header to the `detectLocaleFromHttpHeader()` method, which will analyze the header and return the best matching `Locale` object. Also methods exist which accept a locale identifier or template `Locale` object as a parameter and will return a best match.

Translating text

Translator class

The `\Neos\Flow\I18n\Translator` class is the central place for the translation related functionality. Two translation modes can be used: translating by original label or by ID. `Translator` also supports plural forms and placeholders.

For `translateByOriginalLabel()` you need to provide the original (untranslated, source) message to be used for searching the translated message. It makes view templates more readable.

`translateById()` expects you to provide the systematic ID (like *user.notRegistered*) of a message.

Both methods accept the following optional arguments:

- `arguments` - array of values which will replace corresponding placeholders

- `quantity` - integer or decimal number used for finding the correct plural form
- `sourceName` - name of source catalog to read the translation from.
- `packageKey` of the package the source catalog is contained in.

Hint: Translation by label is very easy and readable, but if you ever want to change the original text, you are in trouble. The use of IDs gives you more flexibility in that respect.

Another issue: some labels do not contain their context, like "Name". What is meant here, a person's name or a category label? This can be solved by using IDs that convey the context (note that both could be "Name" in the final output):

- party.person.fullName
- blog.category.name

We therefore recommend to use `translationById()` in your code.

Plural forms

The `Translator` supports plural forms. English has only two plural forms: *singular* and *plurals* but the CLDR defines six plural forms: *zero*, *one*, *two*, *few*, *many*, *other*. Though english only uses *one* and *other*, different languages use more forms (like *one*, *few*, and *other* for Polish) or less forms (like only *other* for Japanese).

Sets of rules exist for every language defining which plural form should be used for a particular quantity of a noun. If no rules match, the implicit *other* rule is assumed. This is the only form existing in every language.

If the catalogs with translated messages define different translations for particular plural forms, the correct form can be obtained by the `Translator` class. You just need to provide the `quantity` parameter - an integer or decimal number which specifies the quantity of a noun in the sentence being translated.

Placeholders

Translated messages (labels) can contain placeholders - special markers denoting he place where to insert a particular value and optional configuration on how to format it.

The syntax of placeholders is very simple:

```
{id[,formatter[,attribute1[,attribute2...]]]}
```

where:

- *id* is an integer used to index the arguments to insert
- *formatter* (optional) is a name of one of the *Formatters* to use for formatting the argument (if no formatter is given the provided argument will be cast to string)
- *attributes* (optional) are strings directly passed to the `Formatter`. What they do depends on the concrete `Formatter` which is being used, but generally they are used to specify formatting more precisely.

Some examples:

```
{0}
{0,number,decimal}
{1,datetime,time,full}
```

1. The first example would output the first argument (indexing starts with 0), simply string-casted.

2. The second example would use `NumberFormatter` (which would receive one attribute: *decimal*) to format first argument.
3. The third example would output the second argument formatted by the `DatetimeFormatter`, which would receive two attributes: *time* and *full* (they stand for format *type* and *length*, accordingly).

Formatters

A `Formatter` is a class implementing the `\Neos\Flow\I18n\Formatter\FormatterInterface`. A formatter can be used to format a value of particular type: to convert it to string in locale-aware manner. For example, the number *1234.567* would be formatted for French locale as *1 234,567*. It is possible to define more elements than just the position and symbols of separators.

Together with placeholders, formatters provide robust and easy way to place formatted values in strings. But formatters can be used directly (i.e. not in placeholder, but in your class by injection), providing you more control over the results of formatting.

The following formatters are available in Flow by default:

`\Neos\Flow\I18n\Formatter\NumberFormatter` Formats integers or floats in order to display them as strings in localized manner. Uses patterns obtained from CLDR for specified locale (pattern defines such elements like minimal and maximal size of decimal part, symbol for decimal and group separator, etc.). You can indirectly define a pattern by providing format type (first additional attribute in placeholder) as *decimal* or *percent*. You can also manually set the pattern if you use this class directly (i.e. not in placeholder, but in your class by injection).

`\Neos\Flow\I18n\Formatter\DatetimeFormatter` Formats date and / or time part of PHP `\DateTime` object. Supports most of very extensive pattern syntax from CLDR. Has three format types: *date*, *time*, and *datetime*. You can also manually set the pattern if you use this class directly.

The following parameters are generally accepted by Formatters' methods:

- `locale` - formatting result depends on the localization, which is defined by provided `Locale` object
- `formatLength` (optional) - CLDR provides different formats for *full*, *long*, *medium*, *short*, and *default* length

Every formatter provides few methods, one for each format type. For example, `NumberFormatter` has methods `formatDecimalNumber()` - for formatting decimals and integers - and `formatPercentNumber()` - for percentage (parsed value is automatically multiplied by 100).

You can create your own formatter class which will be available for use in placeholders. Just make sure your class implements the `\Neos\Flow\I18n\Formatter\FormatterInterface`. Use the fully qualified class name, without the leading backslash, as formatter name:

```
{0,Acme\Foobar\Formatter\SampleFormatter}
```

Translation Providers

Translation providers are classes implementing the `TranslationProviderInterface`. They are used by the `Translator` class for accessing actual data from translation files (message catalogs).

A `TranslationProvider`'s task is to read (understand) the concrete format of catalogs. Flow comes with one translation provider by default: the `XliffTranslationProvider`. It supports translations stored in *XLIFF message catalogs*, supports plural forms, and both translation modes.

You can create and use your own translation provider which reads the file format you need, like *PO*, *YAML* or even *PHP* arrays. Just implement the interface mentioned earlier and use the *Objects.yaml* configuration file to set your

translation provider to be injected into the `Translator`. Please keep in mind that you have to take care of overrides yourself as this is within the responsibilities of the translation provider.

Fluid ViewHelper

There is a `TranslateViewHelper` for Fluid. It covers all `Translator` features: it supports both translation modes, plural forms, and placeholders. In the simplest case, the `TranslateViewHelper` can be used like this:

```
<f:translate id="label.id"/>
```

It will output the translation with the ID "label.id" (corresponding to the trans-unit id in XLIFF files).

The `TranslateViewHelper` also accepts all optional parameters the `Translator` does.

```
<f:translate id="label.id" source="someLabelsCatalog" arguments="{0: 'foo', 1: '99.9'}
↪"/>
```

It will translate the label using *someLabelsCatalog*. Then it will insert string casted value "*foo*" in place of *{0}* and localized formatted *99.9* in place of *{1,number}*.

Translation by label is also possible:

```
<f:translate>Unregistered User</f:translate>
```

It will output the translation assigned to *user.unregistered* key.

When the translation for particular label or ID is not found, value placed between `<f:translate>` and `</f:translate>` tags will be displayed.

Localizing validation error messages

Flow comes with a bundle of translations for all basic validator error messages. To make use of these translations, you have to adjust your templates to make use of the `TranslateViewHelper`.

```
<f:validation.results for="{property}">
      <f:for each="{validationResults.errors}" as="error">
            {error -> f:translate(id: error.code, arguments: error.arguments,␣
↪package: 'Neos.Flow', source: 'ValidationErrors')}
      </f:for>
</f:validation.results>
```

If you want to change the validation messages, you can use your own package and override the labels there. See the "XLIFF file overrides" section below.

Tip: If you want to have different messages depending on the property, for example if you want to be more elaborate about specific validation errors depending on context, you could add the property to the translate key and provide your own translations.

Localizing resources

Resources can be localized easily in Flow. The only thing you need to do is to put a locale identifier just before the extension. For example, *foobar.png* can be localized as *foobar.en.png*, *foobar.de_DE.png*, and so on. This works with any resource type when working with the Flow ResourceManagement.

Just use the `getLocalizedFilename()` of the i18n `Service` singleton to obtain a localized resource path by providing a path to the non-localized file and a `Locale` object. The method will return a path to the best matching localized version of the file.

Fluid ViewHelper

The `ResourceViewHelper` will by default use locale-specific versions of any resources you ask for. If you want to avoid that you can disable that:

```
{f:uri.resource(path: 'header.png', localize: 0)}
```

Validating and parsing input

Validators

A validator is a class implementing `ValidatorInterface` and is used by the Flow Validation Framework for assuring correctness of user input. Flow provides two validators that utilize i18n functionality:

`\Neos\Flow\Validation\Validator\NumberValidator` Validates decimal and integer numbers provided as strings (e.g. from user's input).

`\Neos\Flow\Validation\Validator\DateTimeValidator` Validates date, time, or both date and time provided as strings.

Both validators accept the following options: *locale*, *strictMode*, *formatType*, *formatLength*.

These validators are working on top of the parsers API. Please refer to the *Parsers* documentation for details about functionality and accepted options.

Parsers

A Parsers' task is to read user input of particular type (e.g. number, date, time), with respect to the localization used and return it in a form that can be further processed. The following parsers are available in Flow:

`\Neos\Flow\I18n\Parser\NumberParser` Accepts strings with integer or decimal number and converts it to a float.

`\Neos\Flow\I18n\Parser\DatetimeParser` Accepts strings with date, time or both date and time and returns an array with date / time elements (like day, hour, timezone, etc.) which were successfully recognized.

The following parameters are generally accepted by parsers' methods:

- *locale* - formatting results depend on the localization, which is defined by the provided `Locale` object
- *formatLength* - CLDR provides different formats for *full*, *long*, *medium*, *short*, and *default* length
- *strictMode* - whether to work in *strict* or *lenient* mode

Parsers are complement to *Formatters*. Every parser provides a few methods, one for each format type. Additionally each parser has a method which accepts a custom format (pattern). You can provide your own pattern and it will be used for matching input. The syntax of patterns depends on particular parser and is the same for a corresponding formatter (e.g. `NumberParser` and `NumberFormatter` support the same pattern syntax).

Parsers can work in two modes: *strict* and *lenient*. In *strict* mode, the parsed value has to conform the pattern exactly (even literals are important). In *lenient* mode, the pattern is only a "base". Everything that can be ignored will be ignored, some simplifications in the pattern are done. The parser tries to do it's best to read the value.

XLIFF message catalogs

The primary source of translations in Flow are XLIFF message catalogs. XLIFF, the *XML Localisation Interchange File Format* is an OASIS-blessed standard format for translations.

Note: In a nutshell an XLIFF document contains one or more `<file>` elements. Each file element usually corresponds to a source (file or database table) and contains the source of the localizable data. Once translated, the corresponding localized data for one, and only one, locale is added.

Localizable data are stored in `<trans-unit>` elements. The `<trans-unit>` contains a `<source>` element to store the source text and a (non-mandatory) `<target>` element to store the translated text.

File locations and naming

Each Flow package may contain any number of XLIFF files. The location for these files is the *Resources/Private/Translations* folder. The files there can be named at will, but keep in mind that *Main* is the default catalog name. The target locale is then added as a directory hierarchy in between. The minimum needed to provide message catalogs for the *en* and *de* locales thus would be:

```
Resources/
  Private/
    Translations/
      en/
        Main.xlf
      de/
        Main.xlf
```

XLIFF file creation

It is possible to create initial translation files for a given language. With Flow command

```
./flow kickstart:translation --package-key Some.Package --source-language-key en --
↪target-language-keys "de,fr"
```

the files for the default language *english* in the package *Some.Package* will be created as well as the translation files for *german* and *french*. Already existing files will not be overwritten. Translations that do not yet exist are generated based on the default language.

A minimal XLIFF file looks like this:

```
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
        <file original="" source-language="da" target-language="fr" datatype=
↪"plaintext">
```

(continues on next page)

(continued from previous page)

```
                <body>
                        <trans-unit id="danish.celebrity">
                                <source>Skarhøj</source>
                                <target>Sarkosh</target>
                        </trans-unit>
                </body>
        </file>
</xliff>
```

If possible you should set up your editor to use the XLIFF 1.2 strict schema to validate the files you are working on.

Note: When using `translationById()` the framework will check the catalog's source language against the currently needed locale and use the `<source>` element if no `<target>` element is found. This eliminates the need to duplicate messages in catalogs where source and target language are the same.

But you may still ask yourself *do I really need to duplicate all the strings in XLIFF files?* The answer is *you should*. Using target allows to fix typos or change wording without breaking translation by label for all other languages.

How to create meaningful XLIFF ids

When using the recommended way of translating by id, it is even more important to use meaningful identifiers. Our suggestion is to group identifiers and use dot notation to build a hierarchy that is meaningful and intuitive:

```
settings.account.keepLoggedIn
settings.display.compactControls
book.title
book.author
...
```

Labels may contain placeholders to be replaced with given arguments during output. Earlier we saw an example use of the TranslateViewHelper:

```
<f:translate id="label.id" arguments="{0: 'foo', 1: '99.9'}"/>
```

The corresponding XLIFF files will contain placeholders in the source and target strings:

```
<trans-unit id="some.label">
        <source>Untranslated {0} and {1,number}</source>
        <target>Übersetzung mit {1,number} und {0}</target>
</trans-unit>
```

As you can see, placeholders may be reordered in translations if needed.

Plural forms in XLIFF files

Plural forms are also supported in XLIFF. The following example defines a string in two forms that will be used depending on the count:

```
<group id="some.label" restype="x-gettext-plurals">
        <trans-unit id="some.label[0]">
                <source>This is only {0} item.</source>
                <target>Dies ist nur {0} Element.</target>
        </trans-unit>
        <trans-unit  id="some.label[1]">
                <source>These are {0} items.</source>
                <target>Dies sind {0} Elemente.</target>
        </trans-unit>
</group>
```

Please be aware that the number of the available plural forms depends on the language! If you want to find out which plural forms are available for a locale you can have a look at *Neos.Flow/Resources/Private/I18n/CLDR/Sources/supplemental/plurals.xml*

XLIFF file translation

To translate XLIFF files you can use any text editor, but translation is a lot easier using one the available translation tools. To name two of them: Virtaal is a free and open-source tool for offline use and Pootle (both from the Translate Toolkit project) is a web-based translation server.

XLIFF can also easily be converted to *PO* file format, edited by well known *PO* editors (like *Poedit*, which supports plural forms), and converted back to *XLIFF* format. The *xliff2po* and *po2xliff* tools from the *Translate Toolkit* project can convert without information loss.

XLIFF file overrides

As of Flow 4.2, XLIFF files are no longer solely identified by their location in the file system. Instead, the `<file>`'s `product-name` and `original` attributes are evaluated to the known `package` and `source` properties, if given. The actual location in the file system is only taken into account if this information is missing and mainly for backwards compatibility.

This allows for an override mechanism, which comes in two levels:

Package-based overrides

Translation files are assembled by collecting labels along the composer dependency graph. This means that as long a package depends (directly or indirectly) on another package, it can override or enrich the other package's XLIFF files by using the other package's `product-name` and `original` values.

Note: If you have trouble overriding another package's translations, please check your `composer.json` if you correctly declared that package as a dependency.

Global translations overrides

In case translations are provided by another source than packages (e.g. via import from a third party system), a global translation path can be declared and is evaluated with highest priority in that it overrides all translations provided by packages. The default value for this is `Data/Translations` and can be changed via the configuration parameter

```
Neos:
  Flow:
    i18n:
      globalTranslationPath: '%FLOW_PATH_DATA%Translations/'
```

Example

`Packages/Framework/Neos.Flow/Resources/Private/Translations/en/ValidationErrors.xlf`

```
<file original="" product-name="Neos.Flow" source-language="en" datatype="plaintext">
  <body>
    <trans-unit id="1221551320" xml:space="preserve">
      <source>Only regular characters (a to z, umlauts, ...) and numbers are allowed.
↪</source>
    </trans-unit>
  </body>
</file>
```

`Packages/Application/Acme.Package/Resources/Private/Translations/en/ValidationErrors.xlf`

```
<file original="ValidationErrors" product-name="Neos.Flow" source-language="en"␣
↪datatype="plaintext">
  <body>
    <trans-unit id="1221551320" xml:space="preserve">
      <source>Whatever translation more appropriate to your domain comes to your mind.
↪</source>
    </trans-unit>
  </body>
</file>
```

Note: In case of undetected labels, please make sure the `original` and `product-name` attributes are properly set (or not at all, if the file resides in the matching directory). Since these fields are used to detect overrides, they are now meaningful and cannot be filled arbitrarily any more.

2.3.20 Error and Exception Handling

Flow reports applications errors by throwing specific exceptions. Exceptions are structured in a hierarchy which is based on base exception classes for each component. By default, PHP catchable errors, warnings and notices are automatically converted into exceptions in order to simplify the error handling.

In case an exception cannot be handled by the application, a central exception handler takes over to display or log the error and shut down the application gracefully.

Throwing Exceptions

Applications should throw exceptions which are based on one of the exception classes provided by Flow. Each exception should be identified by a unique error code which is, by convention, the unix timestamp of the point in time when the developer implemented the code throwing the exception:

```
if ($somethingWentReallyWrong) {
    throw new SomethingWentWrongException('An exception message', 1347145643);
}
```

Exceptions can contain an HTTP status code which is sent as a corresponding response header. The status code is simply set by defining a property with the respective value assigned:

```
class SomethingWasNotFoundException extends \Neos\Flow\Exception {

/**
 * @var integer
 */
protected $statusCode = 404;

}
```

Exception Handlers

Flow comes with two different exception handlers:

- the `DebugExceptionHandler` displays a big amount of background information, including a call stack, in order to simplify debugging of the exception cause. The output might contain sensitive data because method arguments are displayed in the backtrace.
- the `ProductionExceptionHandler` displays a neutral message stating that an error occurred. Apart from a reference code no information about the nature of the exception or any parameters is disclosed.

By default, the `DebugExceptionHandler` is used in Development context and the `ProductionExceptionHandler` is in charge in the Production context.

The exception handler to be used can be configured through an entry in Settings.yaml:

```
Neos:
  Flow:
    error:
      exceptionHandler:
        # Defines the global, last-resort exception handler.
        # The specified class must implement \Neos\Flow\Error\
↪ExceptionHandlerInterface
        className: 'Neos\Error\Messages\ProductionExceptionHandler'
```

Reference Code

In a production context, the exception handler should, for security reasons, not reveal any information about the inner workings and data of the application. In order to be able to track down the root of the problem, Flow generates a unique reference code when an exception is thrown. It is safe to display this reference code to the user who can, in turn, contact the administrators of the application to report the error. At the server side, detailed information about the exception is stored in a file named after the reference code.

You will find report files for exceptions thrown in `Data/Logs/Exceptions/`. In some rare cases though, when Flow is not even able to write the respective log file, no details about the exception can be provided.

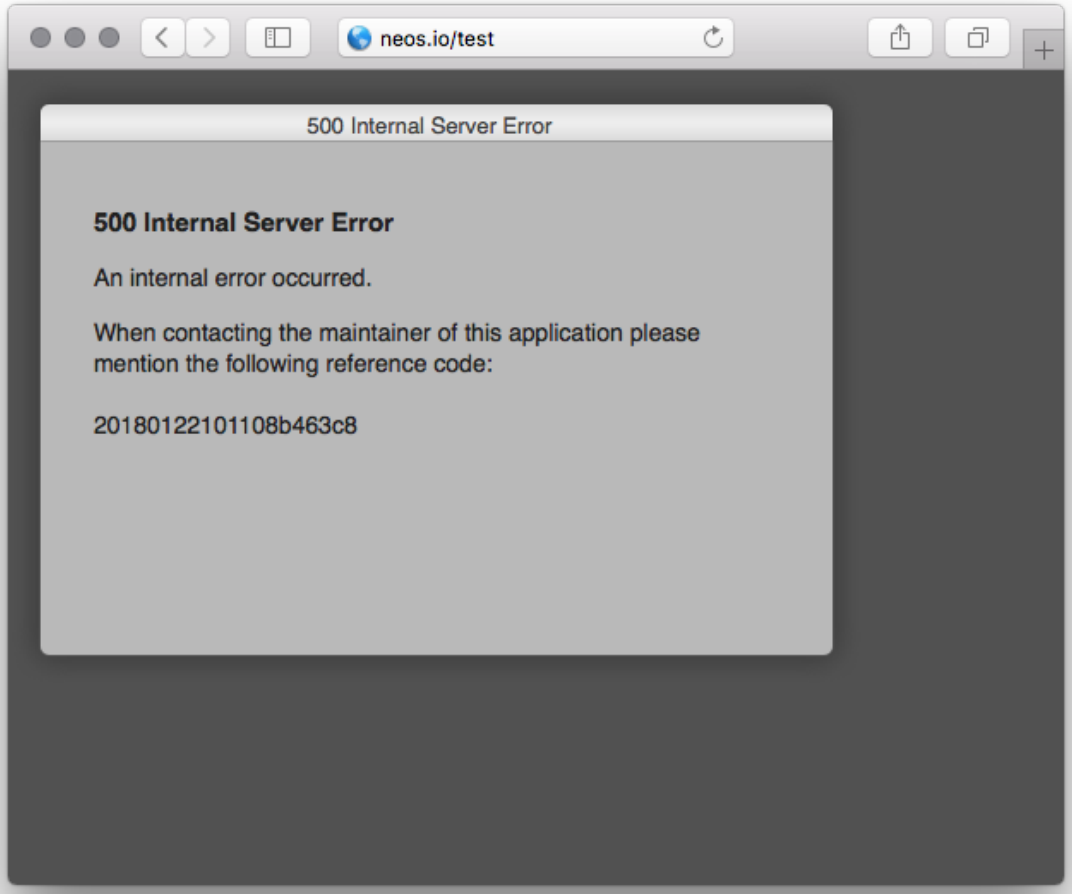

Fig. 34: Exception screen with reference code

Error Handler

Flow provides a central error handler which jumps in if a PHP error, warning or notice occurs. Instead of displaying or logging the error right away, it is transformed into an `ErrorException`.

A configuration option in Settings.yaml allows for deciding which error levels should be converted into exceptions. All other errors are silently ignored:

```
Neos:
  Flow:
    error:
      errorHandler:
        # Defines which errors should result in an exception thrown - all other error
        # levels will be silently ignored. Only errors that can be handled in an
        # user-defined error handler are affected, of course.
        exceptionalErrors: ['%E_USER_ERROR%', '%E_RECOVERABLE_ERROR%']
```

Custom Error Views

In order to allow customized, specifically looking error templates; even depending on the nature of an error; Flow provides configurable rendering groups. Each such rendering group holds information about what template to use, what text information should be provided, and finally, what HTTP status codes or what Exception class names each rendering group is responsible for.

An example configuration could look like in the following Settings.yaml excerpt:

```
Neos:
  Flow:
    error:
      exceptionHandler:
        defaultRenderingOptions: []

        renderingGroups:

          notFoundExceptions:
            matchingStatusCodes: [404]
            options:
              viewClassName: 'Neos\FluidAdaptor\View\StandaloneView'
              viewOptions:
                templatePathAndFilename: 'resource://Neos.Flow/Private/Templates/
↪Error/Default.html'
              variables:
                errorDescription: 'Sorry, the page you requested was not found.'

          databaseConnectionExceptions:
            matchingExceptionClassNames: ['Neos\Flow\Persistence\Doctrine\
↪DatabaseConnectionException']
            options:
              viewClassName: 'Neos\FluidAdaptor\View\StandaloneView'
              viewOptions:
                templatePathAndFilename: 'resource://Neos.Flow/Private/Templates/
↪Error/Default.html'
              variables:
                errorDescription: 'Sorry, the database connection couldn''t be␣
↪established.'
```

defaultRenderingOptions: this carries default options which can be overridden by the `options` key of a particular rendering group; see below.

`notFoundExceptions` and `databaseConnectionExceptions` are freely chosen, descriptive key names, their actual naming has no further implications.

matchingStatusCodes: an array of integer values what HTTP status codes the rendering group is for

matchingExceptionClassNames: an array of string values what Exception types the rendering group is for. Keep in mind that, as always the class name must not contain a leading slash, but must be fully qualified, of course.

`options`:

logException: a boolean telling Flow to log the exception and write a backtrace file. This is on by default but switched off for exceptions with a 404 status code

renderTechnicalDetails: a boolean passed to the error template during rendering and used in the default error template to include more details on the error at hand. Defaults to FALSE but is set to TRUE for development context.

viewClassName: a class name of the view that should be used

viewOptions: an array of options handed to the view. See `$supportedOptions` of the used view

templatePathAndFilename: @deprecated (use `viewOptions.templatePathAndFilename: 'file'`) a resource string to the filename to use

layoutRootPath: @deprecated (use `viewOptions.layoutRootPaths: ['path']`) a resource string to the layout root path

partialRootPath: @deprecated (use `viewOptions.partialRootPaths: ['path']`) a resource string to the partial root path

format: @deprecated the format to use, for example `html` or `json`, if appropriate

variables an array of additional, arbitrary variables which can be accessed in the template

The following variables will be assigned to the template an can be used there:

exception: the Exception object which was thrown

renderingOptions: the complete rendering options array, as defined in the settings. This is a merge of `Neos.Flow.error.exceptionHandler.defaultRenderingOptions` and the `options` array of the particular rendering group

statusCode: the integer value of the HTTP status code which has been thrown (`404`, `503` etc.)

statusMessage: the HTTP status message equivalent, for example `Not Found`, `Service Unavailable` etc. If no matching status message could be found, this value is `Unknown Status`.

referenceCode: the reference code of the exception, if applicable.

2.3.21 Signals and Slots

The concept of *signals* and *slots* has been introduced by the Qt toolkit and allows for easy implementation of the Observer pattern in software.

A *signal*, which contains event information as it makes sense in the case at hand, can be emitted (sent) by any part of the code and is received by one or more *slots*, which can be any function in Flow. Almost no registration, deregistration or invocation code need be written, because Flow automatically generates the needed infrastructure using AOP.

Defining and Using Signals

To define a signal, simply create a method stub which starts with `emit` and annotate it with a `Signal` annotation:

Example: Definition of a signal in PHP

```
/**
 * @param Comment $comment
 * @return void
 * @Flow\Signal
 */
protected function emitCommentCreated(Comment $comment) {}
```

The method signature can be freely defined to fit the needs of the event that is to be signalled. Whatever parameters are defined will be handed over to any slots listening to that signal.

Note: The `Signal` annotation is picked up by the AOP framework and the method is filled with implementation code as needed.

To emit a signal in your code, simply call the signal method whenever it makes sense, like in this example:

Example: Emitting a Signal

```
/**
 * @param Comment $newComment
 * @return void
 */
public function createAction(Comment $newComment) {
        ...
        $this->emitCommentCreated($newComment);
        ...
}
```

The signal will be dispatched to all slots listening to it.

Defining Slots

Basically any method of any class can be used as a slot, even if never written specifically for being a slot. The only requirement is a matching signature between signal and slot, so that the parameters passed to the signal can be handed over to the slot without problems. The following shows a slot, as you can see it differs in no way from any non-slot method.

Example: A method that can be used as a slot

```
/**
 * @param Comment $comment
 * @return void
 */
public function sendNewCommentNotification(Comment $comment) {
        $mail = new \Neos\SwiftMailer\Message();
        $mail->setFrom(array('john@doe.org ' => 'John Doe'))
                ->setTo(array('karsten@neos.io ' => 'Karsten Dambekalns'))
                ->setSubject('New comment')
                ->setBody($comment->getContent())
                ->send();
}
```

Depending on the wiring there might be an extra parameter being given to the slot that contains the class name and method name of the signal emitter, separated by `::`.

Wiring Signals and Slots Together

Which slot is actually listening for which signal is configured ("wired") in the bootstrap code of a package. Any package can of course freely wire its own signals to its own slots, but also wiring any other signal to any other slot is possible. You should be a little careful when wiring your own or even other package's signals to slots in other packages, as the results could be non-obvious to someone using your package.

When Flow initializes, it runs the `boot()` method in a package's `Package` class. This is the place to wire signals to slots as needed for your package:

Example: Wiring signals and slots together

```
/**
 * Boot the package. We wire some signals to slots here.
 *
 * @param \Neos\Flow\Core\Bootstrap $bootstrap The current bootstrap
 * @return void
 */
public function boot(\Neos\Flow\Core\Bootstrap $bootstrap) {
        $dispatcher = $bootstrap->getSignalSlotDispatcher();
        $dispatcher->connect(
                \Some\Package\Controller\CommentController::class, 'commentCreated',
                \Some\Package\Service\Notification::class, 'sendNewCommentNotification
↪'
        );
}
```

The first pair of parameters given to `connect()` identifies the signal you want to wire, the second pair the slot.

The signal is identified by the class name and the signal name, which is the method name without `emit`. In the above example, the method which triggers the `commentCreated` signal is called `emitCommentCreated()`.

The slot is identified by the class name and method name which should be called. If the method name starts with `::` the slot will be called statically.

Note:

- Use the `::class` constant to sepcify the class name
- The signal name is the method name **without** `emit`

An alternative way of specifying the slot is to pass an object instance instead of a class name to the `connect` method. One can also pass a `Closure` instance to react to signals, in this case the slot method name can be omitted:

```
$dispatcher->connect(\Acme\Com\Service::class, 'thingsChanged', function (
↪$changedThings) {
    // do something here
});
```

There is one more parameter available: `$passSignalInformation`. It controls whether or not the signal information (class name and method name of the signal emitter, separated by `::`) should be passed to the slot as last parameter. `$passSignalInformation` is `TRUE` by default.

Note: Slots with a variable number of arguments may use the signal information in unexpected ways. If in doubt, set `$passSignalInformation` to `false`.

2.3.22 Reflection

Reflection describes the practice to retrieve information about a program itself and it's internals during runtime. It usually also allows to modify behavior and properties.

PHP already provides reflection capabilities, using them it's possible to, for example, change the accessibility of properties, e.g. from `protected` to `public`, and access methods even though access to them is restricted.

Additionally it's possible to gain information about what arguments a method expects, and whether these are required or optional.

Reflection in Flow

Flow provides a powerful extension to PHP's own basic reflection functionality, not only adding more capabilities, but also speeding up reflection massively. It makes heavy use of the annotations (tags) found in the documentation blocks, which is another important reason why you should exercise care about a correct formatting and respecting some rules when applying these.

Note: A specific description about these DocComment formatting requirements is available in the *Coding Guidelines*.

The reflection of Flow is handled via the *Reflection Service* which can be injected as usual.

Example: defining and accessing simple reflection information

```
/**
 * This is the description of the class.
 */
class CustomizedGoodsOrder extends AbstractOrder {

        /**
         * @var \Magrathea\Erp\Service\OrderNumberGenerator
         */
        protected $orderNumberGenerator;

        /**
         * @var \DateTime
```

(continues on next page)

(continued from previous page)

```
         */
        protected $timestamp;

        /**
         * The customer who placed this order
         * @var \Magrathea\Erp\Domain\Model\Customer
         */
        protected $customer;

        /**
         * The order number, for example ME-3020-BB
         * @var string
         */
        protected $orderNumber;

        /**
         * @param \Magrathea\Erp\Domain\Model\Customer $customer;
         */
        public function __construct(Customer $customer) {
                $this->timestamp = new \DateTime();
                $this->customer = $customer;
                $this->orderNumber = $this->orderNumberGenerator->createOrderNumber();
        }

        /**
         * @return \Magrathea\Erp\Domain\Model\Customer
         */
        public function getCustomer() {
                return $this->customer;
        }
}
```

In an application, after wiring $reflectionService with `\Neos\Flow\Reflection\ReflectionService` via, for example, Dependency Injection, there are a couple of options available. The following two examples just should give a slight overview.

Listing all sub classes of the `AbstractOrder` class*

```
$this->reflectionService->getAllSubClassNamesForClass('Magrathea\Erp\Domain\Model\
↪AbstractOrder'));
```

returns `array('Magrathea\Erp\Domain\Model\CustomizedGoodsOrder')`.

Fetching the plain annotation tags of the `customer` property from the `CustomizedGoodsOrder` class

```
$this->reflectionService->getPropertyTagsValues('Magrathea\Erp\Domain\Model\
↪CustomizedGoodsOrder', 'customer'));``
```

returns `array('var' => '\Magrathea\Erp\Domain\Model\Customer')`

The API doc of the ReflectionService shows all available methods. Generally said, whatever information is needed to gain information about classes, their properties and methods and their sub or parent classes and interface implementations, can be retrieved via the reflection service.

Custom annotation classes

A powerful feature is the ability to introduce customized annotation classes; this achieves, for example, what across the framework often can be seen with the `@Flow\...` or `@ORM\...` annotations.

Create an annotation class

An annotation class is best created in a direct subdirectory of your `Classes` one and carries the name `Annotations`. The class itself receives the name exactly like the annotation should be.

Example: a ``Reportable`` annotation for use as class and property annotation:

```
<?php
namespace Magrathea\Erp\Annotations;

/**
 * Marks the class or property as reportable, It will then be doing
 * foo and bar, but not quux.
 *
 * @Annotation
 * @Target({"CLASS", "PROPERTY"})
 */
final class Reportable {

        /**
         * The name of the report. (Can be given as anonymous argument.)
         * @var string
         */
        public $reportName;

        /**
         * @param array $values
         */
        public function __construct(array $values) {
                if (!isset($values['value']) && !isset($values['reportName'])) {
                        throw new \InvalidArgumentException('A Reporting annotation
↪must specify a report name.', 1234567890);
                }
                $this->reportName = isset($values['reportName']) ? $values['reportName
↪'] : $values['value'];
        }
}
?>
```

This defines a `Reportable` annotation, with one argument, `reportName`, which is required in this case. It can be given with it's name or *anonymous*, as the sole (and/or first) argument to the value. The annotation can only be used on classes or properties, using it on a method will throw an exception. This is checked by the annotation parser, based on the `Target` annotation. The documentation of the class and it's properties can be used to generate annotation reference documentation, so provide helpful descriptions and names.

Note: An annotation can also be simpler, using only public properties. The use of a constructor allows for some checks and gives the possibility to have *anonymous* arguments, if needed.

This annotation now can be set to arbitrary classes or properties, also across packages. The namespace is introduced using the `use` statement and to shorten the annotation; in the class this annotation can be set to the class itself and to

properties:

```
use Magrathea\Erp\Annotations as ERP;

/**
 * This is the description of the class.
 * @ERP\Reportable(reportName="OrderReport")
 */
class CustomizedGoodsOrder extends AbstractOrder {

        /**
         * @ERP\Reportable
         * @var \Magrathea\Erp\Service\OrderNumberGenerator
         */
        protected $orderNumberGenerator;
```

Accessing annotation classes

With the reflection service, just an instance of your created annotation class is returned, populated with the appropriate information of the annotation itself! So complying with the walkthrough, the following approach is possible:

```
$classAnnotation = $this->reflectionService->getClassAnnotation(
        'Magrathea\Erp\Domain\Model\CustomizedGoodsOrder',
        'Magrathea\Erp\Annotations\Reportable'
);
$classAnnotation instanceof \Magrathea\Erp\Annotations\Reportable;
$classAnnotation->reportName === 'OrderReport';

$propertyAnnotation = $this->reflectionService->getPropertyAnnotation(
        'Magrathea\Erp\Domain\Model\CustomizedGoodsOrder',
        'orderNumberGenerator',
        'Magrathea\Erp\Annotations\Reportable'
);
$propertyAnnotation instanceof \Magrathea\Erp\Annotations\Reportable;
$propertyAnnotation->reportName === NULL;
```

It's even possible to collect all annotation classes of a particular class, done via `reflectionService->getClassAnnotations('Magrathea\Erp\Domain\Model\CustomizedGoodsOrder');` which returns an array of annotations, in this case `Neos\Flow\Annotations\Entity` and our `Magrathea\Erp\Annotations\Reportable`.

2.3.23 Eel

Eel stands for Embedded Expression Language and enables developers to create a Domain Specific Language.

E.g. Neos Fusion has Eel embedded to parse some parts in combination with FlowQuery.

Quickstart

The evaluation consists of two parts, the first one is the expression to evaluate. The second one is the context needed to evaluate the expression.

An expression can be something like:

```
'foo.bar == "Test" || foo.baz == "Test" || reverse(foo).bar == "Test"'
```

To enable this expression to be parsed, a context is needed to define `foo.bar`, `foo.baz` and `reverse()`.

Basically a context is nothing more then an array defining the parts as key value pairs. The above will need a context like:

```
[
    'foo' => [
        'bar' => 'Test1',
        'baz' => 'Test2',
    ],
    'reverse' => function ($array) {
        return array_reverse($array, true);
    },
]
```

To parse the above expression the following code can be used:

```
$expression = 'foo.bar == "Test" || foo.baz == "Test" || reverse(foo).bar == "Test"';
$context = new Context([
    'foo' => [
        'bar' => 'Test1',
        'baz' => 'Test2',
    ],
    'reverse' => function ($array) {
        return array_reverse($array, true);
    }
]);
$result = (new CompilingEvaluator)->evaluate($expression, $context);
```

In the above example `$result` will be a boolean. But the result depends on the expression and can be of any type.

Context Types

Two context types are available.

`Context` will just provide everything you put into the array for the constructor.

`ProtectedContext` will provide the same, except that methods are disallowed by default. You need to explicitly allow methods:

```
$context = new ProtectedContext([
    'String' => new \Neos\Eel\Helper\StringHelper,
]);
$context->allow('String.*');
$result = (new CompilingEvaluator)->evaluate(
    'String.substr("Hello World", 6, 5)',
    $context
);
```

In the above example, all methods for `String` are allowed and therefore the result will be `"World"`.

In case a non allowed method is called, a `\Neos\Eel\NotAllowedException` is thrown.

Evaluators

Two evaluator types are available.

`CompilingEvaluator` will generate PHP Code for expression and cache the expressions.

`InterpretedEvaluator` will not generate PHP Code and evaluate the expression every time. That's useful if you are using Eel outside of Flow context.

Helpers

Helpers provide convenient features like working with math, strings, arrays and dates. Each helper is implemented as a class. No helpers are available out of the box while parsing an expression. To include helpers add them to the context, e.g.

```
$context = new Context([
    'String' => new \Neos\Eel\Helper\StringHelper,
]);
$result = (new CompilingEvaluator)->evaluate(
    'String.substr("Hello World", 6, 5)',
    $context
);
```

The package comes with some predefined helpers you can include in your context. A full, auto generated, list of helpers can be found at Neos Eel Helpers Reference.

Grammar

The full grammar can be found at the Eel repository.

2.3.24 Utility Functions

This chapter contains short introductions to helpful utility functions available in Flow. Please see the API documentation for a full reference:

- `Neos\Utility\ObjectAccess` should be used to get/set properties on objects, arrays and similar structures.
- `Neos\Utility\Arrays` contains some array helper functions for merging arrays or creating them from strings.

- `Neos\Utility\Files` contains functions for manipulating files and directories, and for unifying file access across the different platforms.
- `Neos\Utility\MediaTypes` contains a list of internet media types and their corresponding file types, and can be used to map between them.
- `Neos\Flow\Utility\Now` is a singleton `DateTime` class containing the current time. It should always be used when you need access to the current time.

2.4 Part V: Appendixes

2.5 Contributors

The following is a list of contributors generated from version control information (see below). As such it is neither claiming to be complete nor is the ordering anything but alphabetic.

- Adrian Föder
- Aftab Naveed
- Alexander Berl
- Alex Fruehwirth
- Alexander Kleine-Borger
- Alexander Schnitzler
- Alexander Stehlik
- Andreas Förthner
- Andreas Wolf
- Andy Grunwald
- Aske Ertmann
- Aslambek Idrisov
- Bastian Heist
- Bastian Waidelich
- Benno Weinzierl
- Berit Jensen
- Bernhard Fischer
- Bernhard Schmitt
- Carsten Bleicker
- Carsten Blüm
- Cedric Ziel
- Christian Herberger
- Christian Jul Jensen
- Christian Kuhn
- Christian Müller

- Christian Vette
- Christian Wenzel
- Christoph Daehne
- Christopher Hlubek
- Dan Untenzu
- Daniel Lienert
- Daniel Siepmann
- Daniela Grammlich
- David Kartik
- David Sporer
- David Vogt
- DavidSporer
- Denny Lubitz
- Dmitri Pisarev
- Doehring Daniel
- Dominique Feyer
- Felix Oertel
- Ferdinand Kuhl
- Florian Heinze
- Florian Kaiser
- Franz Kugelmann
- Frederik Löffert
- Fritjof Bohm
- Georg Ringer
- Gerhard Boden
- Hans Hochtl
- Helfer Dominique
- Helmut Hummel
- Henrik Møller Rasmussen
- Ingo Pfennigstorf
- Irene Höppner
- Ivan Litovchenko
- Jacob Floyd
- Jan Hinzmann
- Jan-Erik Revsbech
- Joachim Mathes

- Jochen Rau
- Johannes Hertenstein
- Johannes Künsebeck
- Johannes Steu
- Jon Klixbüll Langeland
- Jon Uhlmann
- Jonas Renggli
- Julian Kleinhans
- Julian Wachholz
- Kai Möller
- Kai Szymanski
- Karol Gusak
- Karsten Dambekalns
- Kay Strobach
- Kerstin Huppenbauer
- Knallcharge
- Lars Lauger
- Lars Peipmann
- Laurent Cherpit
- Leon Kleffmann
- Lienhart Woitok
- Lisa Kampert
- Lorenz Ulrich
- Lucas Krause
- Malte Muth
- Malte Riechmann
- Marc Neuhaus
- Marcin Ryzycki
- Marcin Sągol
- Marco Huber
- Marcos Bjoerkelund
- Markus Goldbeck
- Markus Gunther
- Markus Sommer
- Martin Bless
- Martin Bruggemann

- Martin Ficzel
- Martin Helmich
- Mattias Nilsson
- Max Strübing
- Michael Gerdemann
- Michael Klapper
- Michael Sauter
- Mirjam Bornschein
- Moritz Spindelhirn
- Narongsak Mala
- Nicola Hauke
- Nicolas Hoeller
- Oliver Eglseder
- Oliver Hader
- Pankaj Lele
- Patrick Pussar
- Paul Weiske
- Philipp Maier
- Rafael Kähm
- Raffael Comi
- Ralf Kühnel
- Rens Admiraal
- René Pflamm
- Robert Lemke
- Robin Krahnen
- Roland Waldner
- Roman Minchyn
- Ryan J. Peterson
- Rémy DANIEL
- Salvatore Eckel
- Sascha Egerer
- Sascha Löffler
- Sascha Nowak
- Sebastian Helzle
- Sebastian Heuer
- Sebastian Kurfürst

- Sebastian Sommer
- Simon Schaufelberger
- Simon Schick
- Soren Malling
- Stefan Neufeind
- Steffen Ritter
- Stephan Schuler
- Sören Rohweder
- Søren Malling
- Thomas Blaß
- Thomas Buhk
- Thomas Hempel
- Thomas Layh
- Tim Eilers
- Tim Kandel
- Tim Spiekerkötter
- Tobias Liebig
- Tolleiv Nietsch
- Torsten Blindert
- Troels Thrane
- Tymoteusz Motylewski
- Vaclav Janoch
- Veikko Skurnik
- Wilhelm Behncke
- Wouter Wolters
- WouterJ
- Xavier Perseguers
- Y. Mayer
- Yuri Zaveryukha
- Zach Davis

The list has been generated with some manual tweaking of the output of this:

```
rm contributors.txt
for REPO in `ls` ; do
  cd $REPO
  git log --format='%aN' >> ../contributors.txt
  cd ..
done
```

(continues on next page)

(continued from previous page)

```
sort -u < contributors.txt > contributors-sorted.txt
mv contributors-sorted.txt contributors.txt
```

CHAPTER

THREE

PUBLICATIONS STYLE GUIDE

3.1 About this Guide

The *Publications Style Guide* provides editorial guidelines for text in all kinds of publications, technical documentation, and the software user interface of applications issued by the Neos Project.

Anybody writing text for the Neos Project is encouraged to use this document as a guide to writing style, usage and specific terminology.

3.1.1 Standard Editorial Resources

In general, follow the style and usage rules in:

- Merriam-Webster's Collegiate Dictionary (or other editions by Merriam-Webster)
- The Chicago Manual of Style

3.2 Style and Usage

This chapter provides guidelines on writing style and usage. The intent of these guidelines is to help maintain a consistent voice in publications of the Neos Project and in the user interface.

3.2.1 File Types

Use all caps for abbreviations of file types:

```
a PHP file, a YAML file, the RST file
```

Filename extensions, which indicate the file type, should be in lowercase:

```
.php, .jpg, .css
```

3.2.2 Abbreviations and Acronyms

An *acronym* is a pronounceable word formed from the initial letter or letters of major parts of a compound term. An *abbreviation* is usually formed in the same way but is not pronounced as a word.

Abbreviations are often lowercase or a mix of lowercase and uppercase. Acronyms are almost always all caps, regardless of the capitalization style of the spelled-out form.

- *Latin*: Avoid using Latin abbreviations.
  - *Correct*: `for example, and others, and so on,` and `that is,` or equivalent phrases
  - *Incorrect*: `e.g.` (for example), `et al.` (and others), `etc.` (and so on), `i.e.` (that is)

3.2.3 Above

You can use `above` to describe an element or section of an onscreen document that cannot be paged through (such as a single webpage).

Don't use `above` in print documents; instead, use one of these styles:

- Earlier chapter: Use the chapter name and number:

```
To learn how to create movies, see Chapter 4, "Composing Movies."
```

- Earlier section: Use the section name followed by the page number:

```
For more information, see "Printing" on page 154.
```

- Earlier figure, table, or code listing: Use the number of the element followed by the page number:

```
For a summary of slot and drive numbers, see Table 1-2 (page 36).
```

3.2.4 Braces

Use `braces`, not `curly brackets`, to describe these symbols: { }.

When you need to distinguish between the opening and closing braces, use `left brace` and `right brace`.

3.2.5 Brackets

Use `brackets`, not `squarebrackets`, to describe these symbols: [].

Don't use `brackets` when you mean `angle brackets` (< >).

3.2.6 Capitalization

Three styles of capitalization are available: sentence style, title style, and all caps.

- Sentence-style capitalization:

```
This line provides an example of sentence-style capitalization.
```

- Title-style capitalization:

```
This Line Provides an Example of Title-Style Capitalization.
```

- All caps:

```
THIS LINE PROVIDES AN EXAMPLE OF ALL CAPS.
```

Don't use all caps for emphasis.

3.2.7 Capitalization (Title Style)

Use title-style capitalization for book titles, part titles, chapter titles, section titles (text heads), disc titles, running footers that use chapter titles, and cross-references to such titles.

- **References to specific book elements:** In cross-references to a specific appendix or chapter, capitalize the word Appendix or Chapter (exception to The Chicago Manual of Style). When you refer to appendixes or chapters in general, don't capitalize the word appendix or chapter:

```
See Chapter 2, "QuickTime on the Internet."
See Appendix B for a list of specifications.
See the appendix for specifications.
```

- **References to untitled sections:** In cross-references to sections that never take a title (glossary, index, table of contents, and so on), don't capitalize the name of the section.
- **What to capitalize:** Follow these rules when you use title-style capitalization.

  Capitalize every word except:

  - Articles (`a, an, the`), unless an article is the first word or follows a colon
  - Coordinating conjunctions(`and, but, or, nor, for, yet` and `so`)
  - The word *to* in infinitives (`How to Install Flow`)
  - The word *as*, regardless of the part of speech
  - Words that always begin with a lower case letter, such as `iPad`
  - Prepositions of four letters or fewer (`at, by, for, from, in, into, of, off, on, onto, out, over, to, up` and `with`), except when the word is part of a verb phrase or is used as another part of speech (such as an adverb, adjective, noun, or verb):

```
Starting Up the Computer
Logging In to the Server
Getting Started with Your MacBook Pro
```

Capitalize:

- The first and last word, regardless of the part of speech:

```
For New Mac OS X Users
What the Finder Is For
```

- The second word in a hyphenated compound:

```
Correct: High-Level Events, 32-Bit Addressing
Incorrect: High-level Events, 32-bit Addressing
Exceptions: Built-in, Plug-in
```

- The words `Are, If, Is, It, Than, That` and `This`

3.2.8 Command Line

Write as two separate words when referring to the noun and use the hypenated form `command-line` for and adjective.

3.2.9 Commas

Use a serial comma before `and` or `or` in a list of three or more items.

Correct: `Apple sells MacBook Pro computers, the AirPort Extreme Card, and Final Cut Pro software.`

Incorrect: `Apple sells MacBook Pro computers, the AirPort Extreme Card and Final Cut Pro software.`

3.2.10 Dash (em)

Use the em dash (—) to set off a word or phrase that interrupts or changes the direction of a sentence or to set off a lengthy list that would otherwise make the syntax of a sentence confusing. Don't overuse em dashes. If the text being set off does not come at the end of the sentence, use an em dash both before it and after it:

```
Setting just three edit points--the clip In point, the clip Out point, and the␣
↪sequence In
point--gives you total control of the edit that's performed.
```

To generate an em dash in a reStructured text, use `---`. Close up the em dash with the word before it and the word after it. Consult your department's guidelines for instructions on handling em dashes in HTML.

3.2.11 dash (en)

The en dash (–) is shorter than an em dash and longer than a hyphen. Use the en dash as follows:

- **Numbers in a range:** Use an en dash between numbers that represent the endpoints of a continuous range:

  ```
  bits 3–17, 2003–2005
  ```

- **Compound adjectives:** Use an en dash between the elements of a compound adjective when one of those elements is itself two words:

  ```
  desktop interface–specific instructions
  ```

- **Keyboard shortcuts using combination keystrokes:** Use an en dash between key names in a combination keystroke when at least one of those names is two words or a hyphenated word:

  ```
  Command–Option–Up Arrow, Command–Shift–double-click See also key, keys.
  ```

- **Minus sign:** Use an en dash as a minus sign (except in code font, where you use a hyphen):

  ```
  –1, –65,535
  ```

To generate an en dash in ReStructured Text, use `--`. Close up the en dash with the word (or number) before it and the word (or number) after it.

3.2.12 Kickstarter

A small application provided by the Kickstart paackage, which generates scaffolding for packages, models, controllers and more.

3.3 Font conventions

The following font conventions are used in Flow's documentation:

Italic

Used for URLs, filenames, file extensions, application and package names, emphasis and newly introduced term.

`Monospaced`

Used for class, variable and property names, annotations and other parts of source code which appear in the text.

Command Line

Examples which need to be entered at the command line are shown in a separate text block. Make sure to not type the dollar sign `$` when trying out the commands, as it is the Unix prompt character.

```
$ ./flow kickstart:package Acme.MyPackage
Created .../Acme.Test/Classes/Acme/Test/Controller/
↪StandardController.php
```