FLOW3

Robert Lemke

FLOW3

Robert Lemke

Abstract

FLOW3 is a modern application framework for enterprise-grade PHP applications. This reference describes FLOW3's main features.

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. FLOW3

1. Introduction

FLOW3 is a PHP-based application framework. It is especially well-suited for enterprise-grade applications and explicitly supports Domain-Driven Design, a powerful software design philosophy. Convention over configuration, Test-Driven Development, Continuous Integration and an easy-to-read source code are other important principles we follow for the development of FLOW3.

Although we created FLOW3 as the foundation for the TYPO3 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 FLOW3 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. In fact most of the packages which will be developed for the TYPO3 CMS can be used in any other FLOW3-based application. In essence this reflects the vision of the TYPO3 project: "Inspiring People to Share".

This reference describes all features of FLOW3 and provides you with in-depth information. If you'd like to get a feeling for FLOW3 and get started quickly, we suggest that you try out our Getting Started tutorial first.

> **Note**
>
> Please note that FLOW3 is still under heavy development. Although we hope that the documentation is at least accurate and up to date, it is by no means complete and most likely not proof-read. If you find errors or would like to help us improving the documentation, we're happy to hear from you on our mailing list!

1.1. Overview

The FLOW3 framework is composed of the following submodules:

- The *FLOW3* bootstrap takes care of configuring and initializing the whole framework.
- The *Package* Manager allows you to download, install, configure and uninstall packages.
- The *Object* Manager 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 *Resource* Manager contains functions for providing, caching, securing and retrieving resources.
- The *MVC* Framework takes care of requests and responses and provides you with a powerful, easy-to use Model-View-Controller implementation.
- 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 *Locale* service manages languages and other regional settings and makes them accessible to other packages and FLOW3 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 FLOW3 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 FLOW3 provides and hopefully feel inspired to produce clean and easy-maintainable code.

Tip

A strong coffee helps most people over even the longest documentation.

2. FLOW3 Bootstrap

Note

This section is work in progress and only contains some bullet points for the later documentation.

2.1. Application Context

The FLOW3 Framework can be launched in different application contexts. An application context basically is a set of configuration which has been defined for a certain context. By default, FLOW3 provides configuration for the `Production`, `Development`, `Testing`, and `Staging` context. More contexts may be defined by just adding configuration for it accordingly (refer to the Configuration section to learn more about configuration).

The FLOW3 boot strap (i.e. the class `\F3\FLOW3\FLOW3`) is always instantiated in a single application context. By default (when calling the `index.php` file) the context is `Production`. In the standard distribution a file `index_dev.php` exists which runs FLOW3 in the `Development` context.

2.2. Boot Sequence

At the time of this writing, the sequence in which the various modules of FLOW3 are initialized is hardcoded into the bootstrap. The solution we aim for is, however, a more flexible and cleaner approach which is allows the modules to register themselves for initialization.

3. Packages

FLOW3 is a package-based system. In fact, FLOW3 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. The TYPO3 project [???] hosts a package repository which acts as a convenient hub for interchanging FLOW3 based packages with other community members.

Note

At the time of this writing the package repository for FLOW3 is still in the planning phase.

3.1. Files and Locations

The FLOW3 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 suggested directory layout of a FLOW3 package is as follows:

[PackageName]

- `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 if 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").
- `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. The English manual must be located in a subdirectory called `Manual/en/`. The format for manuals is DocBook [???]. Please refer to the Documentor's Guide for more details about the directories and files within this directory.
- `Meta` — A folder which provides some meta information about the package.
  - `Package.xml` — This mandatory file contains some basic information about the package, such as title, description, author, constraints, version number and more. You should take great care to keep this information updated.
- `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. While public resources will be mirrored into FLOW3's `Public` directory by the Resource Manager (and therefore become accessible from the web) all resources in the `Private` directory remain protected.

`Private`	Contains private resources for the package.
`Public`	Contains private resources for the package.

Although it is up to the package author to name the directories, we suggest the following conventions for directories below `Private` and `Public`:

`Media`	This directory holds images, PDF, Flash, CSS and other files that will be delivered to the client directly without further processing.
`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, ... `;)`

More directories can be added as needed.

`Tests`	Holds the unit tests for the package. Test cases will be recognized by the Testing package if they follow the required naming convention.

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 file name. An example for a valid class name is `\F3\MyPackage\Controller\DefaultController` while the file containing this class would be named `F3_MyPackage_Controller_DefaultController.php`.

All details about naming files, classes, methods and variables correctly can be found in the FLOW3 Coding Guidelines. You're highly encouraged to read (and follow) them.

3.2. 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.

Any public package needs to have a unique package key which you need to register at forge.typo3.org [http://typo3.org] prior to use. But even if you develop a package for private use only, it's clever to register a package key for it.

3.3. Importing and Installing Packages

At this time the features for import and installation of packages have not been implemented. The current behavior is that all directories which are found below the `Packages` folder are assumed to be packages and are active by default. Just make sure that you created a `Package.xml` file in the `Meta` directory of your package.

3.4. Package Manager

The Package Manager is in charge of downloading, installing, configuring and activating packages and registers their objects and resources.

Note

In its current form, the package manager only provides the basic functionality which is necessary to use packages and their objects. More advanced features like installing or configuring packages are of course planned.

3.5. Creating a New Package

Just create the package folder and subdirectories manually and copy & adapt a `Package.xml` file from one of the other packages. Apart from that no further steps are necessary.

3.6. Package Meta Information

All packages need to provide some meta information to the package manager. This data is stored in a file called `Package.xml` which resides in the `Meta` directory of a package. The format of this file follows a RelaxNG schema which is available at http://typo3.org/ns/2008/flow3/package/Package.rng.

Here is an example of a valid `Package.xml` file:

Example 1.1 Package.xml

```
<key>TestPackage</key>
<title>Test Package</title>
<description>The test package to demonstrate the features of Package.xml</de
<version>0.0.1</version>
<state>Alpha</state>
<categories>
   <category>Package.xml</category>
   <category>Testing</category>
</categories>
<parties>
   <person role="LeadDeveloper">
      <name>David Brühlmeier</name>
      <email>typo3@bruehlmeier.com</email>
   </person>
   <person role="Maintainer">
      <name>John Smith</name>
      <email>john@smith.com</email>
      <organisation>Smith Ltd.</organisation>
      <repositoryUserName>jsmith</repositoryUserName>
   </person>
   <organisation role="Sponsor">
      <name>John Doe Co.</name>
      <email>info@johndoe.com</email>
      <website>www.johndoe.com</website>
   </organisation>
</parties>
<constraints>
   <depends>
      <package minVersion="1.0.0" maxVersion="1.9.9">FLOW3</package>
      <system type="PHP" minVersion="5.1.0" />
      <system type="PHPExtension">xml</system>
      <system type="PHPExtension">pgsql</system>
      <system type="PEAR" minVersion="1.5.1">XML_RPC</system>
   </depends>
   <conflicts>
      <system type="OperatingSystem">Windows_NT</system>
   </conflicts>
   <suggests>
      <system type="Memory">16M</system>
   </suggests>
</constraints>

<!-- The following elements are only used and generated by the repository --
<repository>
   <downloads>
      <total>3929</total>
      <thisVersion>444</thisVersion>
   </downloads>
   <uploads>
      <upload>
         <comment>Just a comment...</comment>
         <repositoryUserName>jsmith</repositoryUserName>
         <timestamp>2008-04-22T17:23:09Z</timestamp>
      </upload>
      <upload>
         <comment/>
         <repositoryUserName>jsmith</repositoryUserName>
         <timestamp>2008-04-19T03:54:13Z</timestamp>
      </upload>
   </uploads>
</repository>
</package>
```

Note

If you are working with Eclipse, you might want to install the DEV3 plug-in [http://dev3.org] which - among other tools - provides you with a convenient Package.xml editor

4. 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 FLOW3 and the container.

Tip

A very good start to understand the idea of Inversion of Control and Dependency Injection is reading Martin Fowler's article [http://martinfowler.com/articles/injection.html] on the topic.

4.1. 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 Shipexplained this circumstances nicely in his blog [http://tapestryjava.blogspot.com/2004/08/dependency-injection-mirror-of-garbage.html] (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.

4.1.1. Object Scopes

Objects live in a specific scope. The most commonly used are *prototype* and *singleton:*

Table 1.1. Supported scopes

Scope	Description
singleton (default)	The object instance is unique during one request - each injection by the Object Manager or explicit call of `getObject` returns the same instance. A request can be an HTTP request or a run initiated from the command line.
prototype	The object instance is not unique - each injection or call of the Object Factory's `create` method returns a fresh instance.
session *Not yet implemented*	The object instance is unique during the whole user session - each injection or `getObject` call returns the same instance.

In PHP, objects of the scope prototype are created with the new operator:

```
$myFreshObject = new \F3\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;

 public static 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 FLOW3 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 F3\MyPackage;

/**
 * A sample class
 *
 * @scope prototype
 */
class SomeClass {
}
```

Singleton is the default scope and is therefore assumed if no @scope annotation or other configuration was found.

4.1.2. Creating Prototypes

The instantiation of classes must be handled by the object framework to assert full control over the object lifecycle. In order to instantiate a class or retrieve an existing instance of a class, you'll have to call an API function instead of using the new operator. To create a fresh instance of an object just call the Object Factory's create method:

```
$myFreshObject = $objectFactory->create('F3\MyPackage\MyClassName');
```

The Object Factory (\F3\FLOW3\Object\FactoryInterface) itself is a Singleton and can be acquired like any other object of that scope (see next section).

Analog to new it is possible to pass arguments a constructor of the class being instantiated - they are simply passed as additional arguments to the create method:

```
$myFreshObject = $objectFactory->create('F3\MyPackage\MyClassName', 'first argu
```

4.1.3. 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 1.2. Retrieving the Object Factory through dependency injection

```
namespace F3\MyPackage;

/**
 * A sample class
 */
class SampleClass {

 /**
  * @var \F3\FLOW3\Object\FactoryInterface
  */
 protected $objectFactory;

 /**
  * Constructor.
  * The Object Factory will automatically be passed (injected) by the object fra
  * instantiating this class.
  *
  * @param \F3\FLOW3\Object\FactoryInterface $objectFactory A reference to the
  */
 public function __construct(\F3\FLOW3\Object\FactoryInterface $objectFactory)
  $this->objectFactory = $objectFactory;
 }
}
```

Once the `SampleClass` is being instantiated, the object framework will automagically pass a reference to the Object Factory (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(\F3\FLOW3\Object\ManagerInterface $objectManager) {
 $this->objectManager = $objectManager;
}
```

To explicitly retrieve an object instance use the `getObject()` method:

```
$myObjectInstance = $objectManager->getObject('F3\MyPackage\MyClassName');
```

Like with the `ObjectFactory`'s `create` method, it is possible to pass arguments to the constructor of the object class just by adding them to the `getObject()` call. However passing arguments to a Singleton object makes only sense on the first call when the instance is actually created. On all consecutive calls the arguments are just ignored.

4.1.4. Passing constructor arguments

In most cases an object class will live in the Singleton scope and at most requires a few dependencies passed to its constructor. However, there are times when it becomes necessary to pass dynamic values as constructor arguments, especially when the object

represents an entity and its instances are not unique (Prototype scope). Consider the following classes:

Example 1.3. A simple address book

```
namespace F3\Address;

/**
 * A simple address book
 */
class AddressBook {

 protected $addresses = array();

 public __construct(\F3\iCal\iCalConnectorInterface $iCalConnector) {
  ...
 }

 public addAddress(\F3\Address\Address $address) {
  $this->addresses[] = $address;
 }
}

/**
 * An address
 *
 * @scope prototype
 */
class Address {

 public __construct($street, $zip, $town, $country) {
  ...
 }
}
```

This is admittedly not the fanciest implementation of an address book, but it should demonstrate two things:

- The class `\F3\Address\AddressBook` is supposed to be a Singleton and obviously depends on a third object type `\F3\iCal\iCalConnectorInterface` which is possibly solved by Dependency Injection (will be explained in a later section).
- The class `\F3\Address\Address` represents the address entity and its instances must not be unique - we surely want more than one address. The Address object also expects a few parameters passed to its constructor.

The following code demonstrates how this address book can be used and constructor arguments are passed to the Address entity:

Example 1.4. Passing constructor arguments

```
 # Explicitly fetch a unique instance of the address book (but better use Depen
$myAddressBook = $objectManager->getObject('F3\Address\AddressBook');

 # Create two new addresses and add them to the address book:
$newAddress = $objectFactory->create('F3\Address\Address', 'Tryggevældevej', '2
$myAddresbook->addAddress($newAddress);

$newAddress = $objectFactory->create('F3\Address\Address', 'An den Brodbänken',
$myAddresbook->addAddress($newAddress);
```

4.1.5. 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
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 initialization method is called. The name of this method is configurable and it is called regardless of whether any dependencies have been injected or not
4. During the life of an object no special lifecycle methods are called
5. On disposal, the function `__destruct()` is called (by PHP itself)

As you can see from the above list, there is only one special method which is not provided by PHP's own means and that is the initialization method. Here's a simple example:

Example 1.5. A object class with an initialization method

```
class Foo {

 protected $bar;
 protected $identifier = 'Untitled';

 public function injectBar(\F3\MyPackage\BarInterface $bar) {
  $this->bar = $bar;
 }

 public function setIdentifier($identifier) {
  $this->identifier = $identifier;
 }

 public function intializeObject() {
  echo ('This object has been initialized.);
 }
}
```

4.2. Object Registration and API

4.2.1. 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 the FLOW3 package or in unit tests 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 `F3\FLOW3\Object\ManagerInterface`.

4.2.2. Object names and types

By default, the name of an object is identical to the PHP class which contains the object's code. A class called `F3\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.

If the object name equals the name of a PHP interface, it is often referred to as a *object type*. An interface called `F3\MyPackage\MyInterface` will be available as an object of the same name as long as there exists one class implementing that interface. Object types can be created and retrieved like regular objects:

```
$objectTypeInstance = $objectFactory->create('F3\SomePackage\SomeInterfaceName'
```

If exactly one class implements the `F3\SomePackage\SomeInterfaceName` interface, `$otherObjectInstance` will contain an instance of that class. If zero or more than one class implements the interface, the Object Factory will throw an exception.

The advantage of using object types instead of regular object names 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.

4.3. 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 FLOW3 framework.

4.3.1. 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 it becomes a nightmare testing them.

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 [http://en.wikipedia.org/wiki/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 (e.g. the `F3\FLOW3\Log\Logger\BackendInterface`). The object itself has no control over what kind of logger backend (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 backend and inject it to the authentication object.

> **Tip**
>
> An article [http://www.ddj.com/dept/java/184405016] 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 section about configuring objects) or they can be solved automatically (so called autowiring). Generally there are two modes of dependency injection supported by FLOW3: *Constructor Injection* and *Setter Injection*.

4.3.1.1. 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 1.6. A simple example for Constructor Injection

```
namespace F3\MyPackage;

class Foo {

 protected $bar;

 public function __construct(\F3\MyPackage\BarInterface $bar) {
  $this->bar = $bar;
 }

 public function doSomething() {
  $this->bar->doSomethingElse();
 }
}
```

So far there's nothing special about this class, it just makes sure that an instance of a class implementing the `\F3\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 FLOW3 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. `\F3\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 1.7. Objects.yaml file for Constructor Injection

```
F3\MyPackage\Foo:
  arguments:
    1: { object: F3\MyPackage\Bar }
```

The three lines above define that an object instance of `\F3\MyPackage\Bar` must be passed to the first argument of the constructor when an instance of the object `F3\MyPackage\Foo` is created.

4.3.1.2. 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 1.8. A simple example for Setter Injection

```
namespace F3\MyPackage;

class Foo {

 protected $bar;

 public function setBar(\F3\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 authentication 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 1.9. Objects.yaml file for Setter Injection

```
F3\MyPackage\Foo:
  properties:
    bar: { object: F3\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, FLOW3 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 1.10. The preferred way of Setter Injection, using an inject method

```
namespace F3\MyPackage;

class Foo {

 protected $bar;

 public function injectBar(\F3\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 (but possible). Using `inject*` methods is the preferred way for setter injection in FLOW3.

Note

If both, a `set*` and a `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 [http://www.springframework.org] for example prefer Setter Injection for its flexibility. The more puristic developers of PicoContainer [www.picocontainer.org] 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 (FLOW3's action controller is such a case). 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

4.3.1.3. Settings Injection

No, this headline is not misspelled. FLOW3 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 1.11. Example for the magic injectSettings method

```
namespace F3\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.

4.3.2. Required and Optional Dependencies

All dependencies defined in a constructor are, by its nature, required. If a dependency can't be solved by autowiring or by configuration, FLOW3's object builder will throw an exception.

Also *autowired setter-injected dependencies* are, by default, required. There is a way to declare a setter-injected dependency as optional without the need to configure the dependency in a Objects configuration file. If an optional dependency can't be solved, it just won't be injected and it is the developer's responsibility to test for the availability of the desired object. FLOW3 uses the @optional annotation for this purpose:

Example 1.12. Marking a setter-injected dependency as optional

```
namespace F3\MyPackage;

/**
 * A very fooish class
 */
class Foo {

 /**
  * @var \F3\MyPackage\BarInterface
  */
 protected $bar;

 /**
  * Injects a bar-ish object
  *
  * @param \F3\MyPackage\BarInterface $bar a kind of Bar object
  * @return void
  * @optional
  */
 public function injectBar(\F3\MyPackage\BarInterface $bar) {
  $this->bar = $bar;
 }

 /**
  * A method which does something
  *
  * @return void
  */
 public function doSomething() {
  $this->bar->doSomethingElse();
 }
}
```

Due to the @optional annotation, the injection of a Bar object is now no longer required. If the object builder can't autowire an object for this injection method, it will now no longer throw an exception.

4.3.3. 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 a 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, FLOW3's policy is not to allow circular dependencies at all. As a workaround you can use setter injection instead of Constructor Injection for either one or both of the objects causing the trouble.

4.4. 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:

1. If they exist, the *`PackageName`*`/Configuration/Objects.*` will be included.
2. Additional configuration defined in the global `Configuration/` 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

4.4.1. Configuration Sources

As already mentioned, the configuration for each object is compiled from different sources. The `Objects.yaml` file is the recommended format and is therefore used in most of the examples. However, the names of the configuration options and their possible values are identical to all configuration sources.

4.4.1.1. 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 FLOW3's general rules for YAML-based configuration.

Example 1.13. Sample Objects.yaml file

```
#                                                                        #
# Object Configuration for the MyPackage package                         #
#                                                                        #

# @package MyPackage
# @version $Id: Objects.yaml 123 2009-01-01 12:00:00Z robert $

F3\MyPackage\Foo:
  arguments:
    1: { object: F3\MyPackage\Baz }
    2: { value: "some string" }
    3: { value: false }
  properties:
    bar: { object: F3\MyPackage\BarInterface }
    enableCache: { setting: MyPackage.Cache.enable }
```

4.4.1.2. Objects.php

As an alternative to YAML, it is possible to write configuration files in plain PHP. However, the PHP file should stick to FLOW3's general rules for PHP-based configuration.

The following code again adds the same configuration as in the Constructor Injection example:

Example 1.14. Sample Objects.php file

```
<?php
declare(ENCODING = 'utf-8');

/*                                                                        *
 * Object Configuration for the MyPackage package                         *
 * (this package doesn't really exist, and even if so, the configuration  *
 * would probably be different)                                           *
 *                                                                        */

/**
 * @package MyPackage
 * @version $Id: Objects.php 123 2009-01-01 12:00:00Z robert $
 */

$c['F3\MyPackage\Foo']->arguments->array(
 1 => array('object' => 'F3\MyPackage\Baz'),
 2 => array('value' => 'some string')
);

 // Demonstrates an alternative syntax:
$c['F3\MyPackage\Foo']->arguments->3->value = FALSE;

$c['F3\MyPackage\Foo']->properties->bar->object = 'F3\MyPackage\BarInterface';
$c['F3\MyPackage\Foo']->properties->enableCache->setting = 'MyPackage.Cache.ena

?>
```

> **Caution**
>
> Only use these files for configuration, for example don't register autoloader methods in the `Objects.php` as this code must be invoked in an earlier stage.

4.4.1.3. 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` is the only supported annotation.

It's up to you defining the scope in the class directly or doing it in a `Objects` 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 1.15. Sample scope annotation

```
/**
 * This is my great class.
 *
 * @scope prototype
 */
class SomeClass {

}
```

4.4.2. 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 [http://www.php.net/manual/en/language.oop5.interfaces.php] 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 1.16. A simple Greeter class

```
namespace F3\MyPackage;

class Greeter {
 public function sayHelloTo($name) {
  echo('Hello ' . $name);
 }
}
```

During initialization the above class will automatically be registered as the object `F3\MyPackage\Greeter` and is available to other objects. In the class code of another object you might find these lines:

Example 1.17. Code using the object F3_MyPackage_Greeter

```
   // Use setter injection for fetching an instance of the \F3\MyPackage\Greete
public function injectGreeter(\F3\MyPackage\Greeter $greeter) {
 $this->greeter = $greeter;
}

public function someAction() {
 $greeter->sayHelloTo('Heike');
}
```

Great, that looks all fine and dandy but what if we want to use the much better object `\F3\OtherPackage\GreeterWithCompliments`? Well, you just configure the object `\F3\MyPackage\Greeter` to use a different class:

Example 1.18. Objects.yaml file for object replacement

```
   # Change the name of the class which represents the object "F3\MyPackage\Gre
F3\MyPackage\Greeter: className: F3\OtherPackage\GreeterWithCompliments
```

Now all objects who ask for a traditional greeter will get the more polite version. However, there comes a sour note with the above example: We can't be sure that the `GreeterWithCompliments` class really provides the necessary `sayHello()` method. The solution is to let both implementations implement the same interface:

Example 1.19. The Greeter object type

```
namespace F3\MyPackage;

interface GreeterInterface {
 public function sayHelloTo($name);
}

class Greeter implements \F3\MyPackage\GreeterInterface {
 public function sayHelloTo($name) {
  echo('Hello ' . $name);
 }
}

namespace F3\OtherPackage;

class GreeterWithCompliments implements \F3\MyPackage\GreeterInterface{
 public function sayHelloTo($name) {
  echo('Hello ' . $name . '! You look so great!');
 }
}
```

Instead of referring to the original implementation directly we can now refer to the interface. In this case we call the object name a *object type* because it contains the name of a PHP interface.

Example 1.20. Code using the object type F3\MyPackage \GreeterInterface

```
   // Use setter injection for fetching an instance of the \F3\MyPackage\Greete
public function injectGreeter(\F3\MyPackage\GreeterInterface $greeter) {
 $this->greeter = $greeter;
}

public function someAction() {
 $greeter->sayHelloTo('Heike');
}
```

Finally we have to set which implementation of the `F3\MyPackage\GreeterInterface` should be active:

Example 1.21. Objects.yaml file for object type definition

```
F3\MyPackage\GreeterInterface: className: F3\OtherPackage\GreeterWithCompliment
```

4.4.3. 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.

4.4.3.1. 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*: name of an objects (or object type) 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.*` files. A path separated by dots "." specifies which setting to inject.

4.4.3.2. Constructor Injection

Arguments for constructor injection are defined through the `arguments` option. Each argument is identified by its position, counting starts with 1.

Example 1.22. Sample class for Constructor Injection

```
namespace F3\MyPackage;

class Foo {

 protected $bar;
 protected $identifier;
 protected $enableCache;

 public function __construct(\F3\MyPackage\BarInterface $bar, $identifier, $enab
  $this->bar = $bar;
  $this->identifier = $identifier;
  $this->enableCache = $enableCache;
 }

 public function doSomething() {
  $this->bar->doSomethingElse();
 }
}
```

Example 1.23. Sample configuration for Constructor Injection

```
F3\MyPackage\Foo:
  arguments:
    1: { object: F3\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 convent to just declare the type of the constructor arguments (like `F3\MyPackage\BarInterface` in the above example) and let the autowiring feature configure and resolve the dependencies for you.

4.4.3.3. Setter Injection

The following class and the related `Objects.yaml` file demonstrate the syntax for the definition of setter injection:

Example 1.24. Sample class for Setter Injection

```
namespace F3\MyPackage;

class Foo {

 protected $bar;
 protected $identifier = 'Untitled';
 protected $enableCache = FALSE;

 public function injectBar(\F3\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 1.25. Sample configuration for Setter Injection

```
F3\MyPackage\Foo:
  properties:
    bar: { object: F3\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.

4.4.3.4. 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 1.26. Injecting an object specified in the settings

```
F3\MyPackage\Foo:
  properties:
    bar: { object: MyPackage.fooStuff.barImplementation }
```

Example 1.27. Example Settings.yaml of MyPackage

```
fooStuff:
  barImplementation: F3\MyPackage\Bars\ASpecialBar
```

4.4.3.5. 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 FLOW3 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 1.28. Nesting object configuration

```
F3\MyPackage\Controller\DefaultController:
  properties:
    cache:
      object:
        name: F3\FLOW3\Cache\VariableCache
        arguments:
          1: value: MyCache
          2:
            object:
              name: F3\FLOW3\Cache\Backend\File
              properties:
                cacheDirectory: value: /tmp/
```

4.4.3.6. Disabling Autowiring

Injecting dependencies is a common task. Because FLOW3 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 1.29. Turning off autowiring support in Objects.yaml

```
F3\MyPackage\MyObject:
  autoWiringMode: off;
```

4.4.4. Custom Factories

Complex objects might require a custom factory which takes care of all important settings and dependencies. As we have seen previously, a cache consists of a frontend, a backend and configuration options for that backend. Instead of creating and configuring these objects on your own, you can use the `F3\FLOW3\Cache\Factory` which provides a convenient `create` method taking care of all the rest.

```
$myCache = $cacheFactory->create('MyCache', 'F3\FLOW3\Cache\VariableCache', 'F3
```

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 1.30. Sample configuration for a Custom Factory

```
F3\FLOW3\Cache\CacheInterface:
  factoryClassName: F3\FLOW3\Cache\Factory
  factoryMethodName: create
```

From now on the Cache Factory's `create` method will be called each time an object of type `CacheInterface` needs to be instantiated. If arguments were passed to the `getObject` or `create` method, they will be passed through to the custom factory method:

Example 1.31. YAML configuration for a Custom Factory with default arguments

```
F3\FLOW3\Cache\CacheInterface:
  factoryClassName: F3\FLOW3\Cache\Factory
  arguments:
    2: value: F3\FLOW3\Cache\VariableCache
    3: value: F3\FLOW3\Cache\Backend\File
    4: value: { cacheDirectory: /tmp }
```

Example 1.32. PHP code using the custom factory

```
$myCache = $objectFactory->create('MyCache');
```

`$objectFactory` is a reference to the `F3\FLOW3\Object\Factory`. The argument with the value `MyCache` is passed to the Cache Factory as the first parameter. The required second and third argument and the optional fourth parameter are automatically built from the values defined in the object configuration.

4.4.5. Name of Lifecycle Method

The default name of a lifecycle method is `initializeObject`. If such a method exists, it will be called after the object has been instantiated and all dependencies are injected. The name of the initialization method 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 1.33. Objects.yaml configuration of the initialization method

```
F3\MyPackage\MyClass:
  lifecycleInitializationMethod: myInitializeMethodname
```

5. Configuration Framework

Configuration is an important aspect of versatile applications. FLOW3 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.

5.1. Configuration Files

FLOW3 distinguishes between different types of configuration. The most important type of configuration are the settings, however other configuration types exist for special purposes.

The preferred configuration format is PHP 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 usually control the visible behaviour of the application.

`FLOW3.yaml` Contains settings for FLOW3 itself.

`Routes.yaml` Contains routes configuration. This routing information is parsed and used by the MVC Web Routing mechanism. Refer to the MVC section for more information.

`Objects.yaml` Contains object configuration, ie. options which configure objects and the combination of those on a lower level. See the Object Manager section for more information.

`Packages.yaml` Contains package configuration, ie. options which define certain specialities of the package such as custom autoloaders or special resources.

5.1.1. 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):

`/Packages/`*`PackageName`*`/Configuration/` The `Configuration` directory of each package is scanned first. Only at this stage new configuration options can be introduced (by just defining a default value). After all configuration files form these directories have been parsed, the resulting configuration containers are protected against further introduction of new options.

Note that `FLOW3.yaml` files are only allowed in the FLOW3 package and are ignored in any other package configuration dir.

`/Configuration/` Configuration in the global `Configuration` directory override the default settings which were defined in the package's configuration directories. To safe users from typos, options which are introduced on this level will result in an error message.

`/Configuration/`*`ApplicationContext`*`/` There may exist a subdirectory for each application context (see FLOW3 Bootstrap section). This configuration is only loaded if FLOW3 runs in the respective application context. Like in the global `Configuration` directory, no new configuration options can be introduced at this point - only their values can be changed.

5.2. Defining Configuration

5.2.1. Configuration Formats

Although YAML is the preferred and default configuration format, it is possible to use other kinds of markup. Currently FLOW3 has built-in support for YAML and PHP based configuration.

5.2.2. YAML

YAML is a well-readable format which is espcially 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 FLOW3, 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 FLOW3 distribution for getting more examples.

Example 1.34. Example for a package-level Settings.yaml

```
#                                                                             #
# Settings Configuration for the TYPO3CR Package                              #
#                                                                             #

# $Id: Settings.yaml 1459 2008-11-10 10:11:28Z k-fish $

TYPO3CR:

  # The storage backend configuration
  storage:
    backend: 'F3\TYPO3CR\Storage\Backend\PDO'
    backendOptions:
      dataSourceName: 'sqlite:%FLOW3_PATH_DATA%Persistent/TYPO3CR.db'
      username:
      password:

  # The indexing/search backend configuration
  search:
    backend: 'F3\TYPO3CR\Storage\Search\Lucene'
    backendOptions:
      indexLocation: '%FLOW3_PATH_DATA%Persistent/Index/'
```

5.2.3. PHP

Although configuration files are plain PHP, they should follow FLOW3's conventions for configuration files. All options are properties of a special configuration container object which is automagically provided in a variable named $c.

Example 1.35. Example for a package-level Settings.php

```
<?php
declare(ENCODING="utf-8");

/*                                                                        *
 * Settings Configuration for the TYPO3CR Package                         *
 *                                                                        */

/**
 * @package TYPO3CR
 * @version $Id: Objects.php 888 2008-05-30 16:00:05Z k-fish $
 */

/**
 * The storage backend to use for TYPO3CR.
 *
 * @var F3_TYPO3CR_Storage_BackendInterface
 */
$c->TYPO3CR->storage->backend = 'F3\TYPO3CR\Storage\Backend\PDO';

/**
 * Options which are passed to the storage backend used by TYPO3CR
 *
 * @var array
 */
$c->TYPO3CR->storage->backendOptions = array(
  'dataSourceName' => 'sqlite:/tmp/TYPO3CR.db',
  'username' => NULL,
  'password' => NULL
);

?>
```

As you can see, you can introduce new configuration options by just assigning a value to them using PHP's object / member variable syntax.

Technically the `$c` variable is an instance of `\F3\FLOW3\Configuration\Container` which automatically creates sub containers as soon as you try to access a not-yet-existing property. Taking the above example, `$c`, `$c->TYPO3CR` and `$c->TYPO3CR->storage` are all Configuration Container objects which were created on the fly. backend is just a property (member variable) of `$c->TYPO3CR->storage` and contains a value of the type string.

Note

Please note that the first level of your option tree must always contain the key of the package you're referring to (TYPO3CR in the above example). While it technically is possible to define and modify settings of other package in your own package, it's certainly something you should do only if really necessary.

There are two alternatives to the above object-driven syntax and all the three of them may be mixed as you like.

Instead of the object / property way, you can equally use an array syntax:

Example 1.36. Settings declaration using the object and the array syntax

```
/**
 * The storage backend to use for TYPO3CR.
 *
 * @var \F3\TYPO3CR\Storage\BackendInterface
 */
$c->TYPO3CR->storage['backend'] = 'F3\TYPO3CR\Storage\Backend\PDO';
```

Finally you may even call virtual setter methods in order to get a more readable configuration file when setting many options on the same container:

Example 1.37. Settings declaration with the object syntax and virtual setters

```
   // Traditional object / property syntax:

$c->Demo->administrator->name = 'John Doe';
$c->Demo->administrator->email = 'johndoe@typo3.org';
$c->Demo->administrator->country = 'DE';

   // The same with the virtual setters syntax:

$c->Demo->administrator->
   setName('JohnDoe')->
   setEmail('johndoe@typo3.org')->
   setCountry('DE');

?>
```

5.2.4. Constants (special case)

In the `Packages` configuration you can use two special options influencing the class autoloader of FLOW3's resource manager.

> **Caution**
>
> This might be a temporary solution which will surely change as soon as the resource manager and the package manager become more mature.

$c->PackageKey->resourceManager->specialClassNameAndPaths	This can be used to register classes outside the include_path and not covered by the FLOW3 class autloader.
$c->PackageKey->resourceManager->includePaths	This can be used to add paths to the PHP include_path.

The values for those options may contain markers which are replaced before the value is used:

%PATH_PACKAGE%	Will be replaced by the path to the package folder.
%PATH_PACKAGE_CLASSES%	Will be replaced by the path to the package's `Classes` folder.
%PATH_PACKAGE_RESOURCES%	Will be replaced by the path to the package's `Resources` folder.

Example 1.38. Example for using specialClassNameAndPaths

```
#                                                                             #
# Package configuration of the Smarty package                                 #
#                                                                             #

# @package Smarty

Smarty:
  resourceManager:
    specialClassNameAndPaths:
      Smarty: %PATH_PACKAGE_RESOURCES%PHP/Smarty/Smarty.class.php
```

Example 1.39. Example for using specialClassNameAndPaths

```
<?php
declare(ENCODING="utf-8");

/*                                                                             *
 * Packages configuration of the Smarty package                                *
 *                                                                             */

/**
 * @package Smarty
 * @version $Id:PackageConfiguration.php 178 2007-03-09 10:30:04Z robert $
 */

$c->Smarty->resourceManager->specialClassNameAndPaths->Smarty = '%PATH_PACKAGE_

?>
```

5.3. Accessing Configuration

There are certain situations in which FLOW3 will automatically provide you with the right configuration - the MVC's Action Controller is such a case. However, in most other cases you will have to retrieve the configuration yourself. The Configuration Manager comes up with a very simple API providing you access to the already parsed and cascaded configuration.

5.3.1. Working with Settings

What you usually want to work with are settings. The following example demonstrates how to let FLOW3 inject an instance of the configuration manager into your class, use it to retrieve the settings for your package and output some option value:

Example 1.40. Retrieving settings

```
class SomeClass {

   protected $configurationManager;

   public function injectConfigurationManager(\F3\FLOW3\Configuration\Manager $
      $this->configurationManager = $configurationManager;
   }

   public function theMethod() {
      $mySettings = $this->configurationManager->getSettings('Demo');
      echo ($mySettings->administrator->name);
      echo ($mySettings->administrator->email);
   }
}
```

5.3.2. Working with other configuration

Although seldomly necessary, it is also possible to retrieve options of the more special configuration types. The configuration manager provides a method called `getSpecialConfiguration()` for this purpose. The result this method returns depends on the actual configuration type you are requesting.

Bottomline is that you should be highly aware of what you're doing when working with these special options. Usually there are much better ways to get the desired information (eg. ask the Object Manager for object configuration).

6. Resource Manager

7. MVC Framework

7.1. Introduction

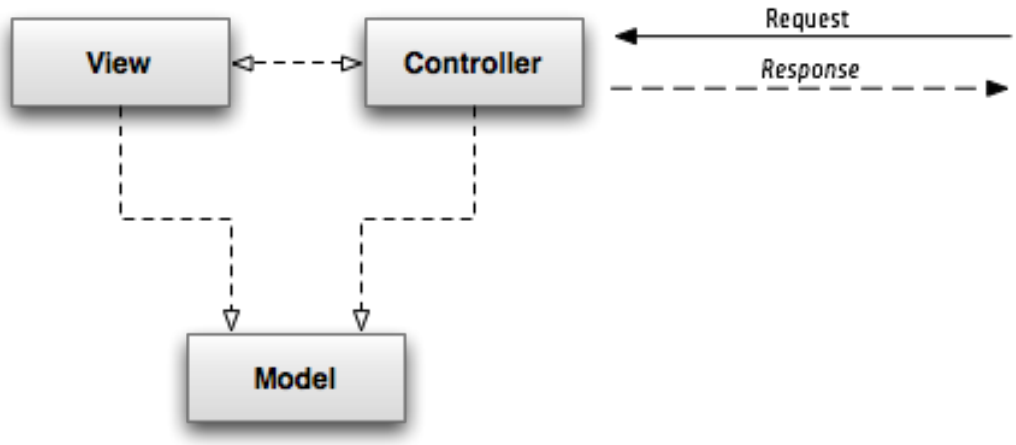
7.1.1. Model-View-Controller

In the design of FLOW3's architecture we have taken great care to separate concerns and assign each part of the framework with well-defined tasks. The separation of concerns is an important principle of good software design and its most prominent representative probably is the Model-View-Controller pattern. MVC separates the business logic from the presentation by splitting up user interaction into three roles:

- The *model* is an object which contains data and business logic of a certain domain. It doesn't contain any information about the presentation of that data, but rather defines the behaviour. In the FLOW3 project we prefer a special kind of model, the Domain Model [http://martinfowler.com/eaaCatalog/domainModel.html].
- The *view* represents the display of the model on the web or another output channel. Views only display data, they don't build or modify it.
- The *controller* reacts on user input, selects and manipulates the model as accordingly, selects a view and passes it the prepared model for rendering.

This diagram outlines the collaboration between model, view and controller:

Figure 1.1. Model-View-Controller Pattern



7.1.2. Other Patterns Used

Design Patterns (and MVC is one of them) are not only great for solving reoccuring design problems in a structured manner - they also help you communicating software designs. The following patterns play an important role in FLOW3's MVC mechanism and might give you a better idea of the overall design:

- Incoming requests are handled by a Request Handler which takes the role of a Front Controller [???].
- Template View [???] is the most commonly used pattern for views, but Transform Views [???] and Two-Step Views [???] are equally supported.
- The preferred type of model is the Domain Model [???].

7.1.3. Hello World!

Let's start with an example before we go into greater detail of request handling and the internals of the MVC framework. The minimal approach is to create an Action Controller which just returns “Hello World!”. To begin with, we need to create some directories which contain the code of our FLOW3 package and eventually the controller class:

```
Packages/
  Demo/
    Classes/
      Controller/
        F3_Demo_Controller_DefaultController.php
```

The DefaultController class looks as simple as this (leaving out the very recommended comments):

Example 1.41. Hello World! controller

```
namespace F3\Demo\Controller;

class DefaultController extends \F3\FLOW3\MVC\Controller\ActionController {
   public function indexAction() {
      return "Hello World!";
   }
}
```

Provided that the web root directory of your local server points to FLOW3's `public/` directory, you will get the following output when calling the URI `http://localhost/demo/` :

```
Hello World!
```

Great, that was easy - but didn't we say that it's the view's responsibility to take care of the presentation? Let's create a simple PHP-based view for that purpose:

```
Packages/
  Demo/
    Classes/
      Controller/
        F3_Demo_Controller_DefaultController.php
      View/
        F3_Demo_View_DefaultIndex.php
```

The view's code is equally trivial:

Example 1.42. Hello World! view

```
namespace F3\Demo\View;

class DefaultIndex extends \F3\FLOW3\MVC\View\AbstractView {
   public function render() {
      return "Hello World!";
   }
}
```

Finally our action controller needs a little tweak to return the rendered view instead of shouting "Hello World!" itself:

Example 1.43. Improved Hello World! controller

```
namespace F3\Demo\Controller;

class DefaultController extends \F3\FLOW3\MVC\Controller\ActionController {
   public function indexAction() {
      return $this->view->render();
   }
}
```

7.1.4. Recommended File Structure

As you have seen in the hello world example, conventions for the directory layout simplify your development a lot. There's no need to register controllers, actions or views if you follow our recommended file structure. These are the rules:

- *Controllers* are located in their own directory `Controller` just below the `Classes` directory of your package. They can have arbitrary names while the `DefaultController` has a special meaning: If the package was specified in the request but no controller, the `DefaultController` will be used.
- *View* classes are situated below a `View` directory. The classname of the view is a combination of the name of the controller and the name of the action.

This sample directory layout demonstrates the above rules:

Example 1.44. Sample file structure

```
Packages/
  Demo/
    Classes/
      Controller/
        F3_Demo_Controller_DefaultController.php
        F3_Demo_Controller_CustomerController.php
        F3_Demo_Controller_OrderController.php
      View/
        F3_Demo_View_DefaultIndex.php
        F3_Demo_View_CustomerIndex.php
        F3_Demo_View_CustomerList.php
        F3_Demo_View_CustomerDetails.php
        F3_Demo_View_OrderList.php
```

Adhering to these conventions has the advantage that the classname of the view for example is resolved automatically. However it is possible (and not really difficult) to deviate from this layout and have a completely different structure.

7.1.5. From the URI to the view

Caution

For the example URIs we assume that the web root directory of your local server points to FLOW3's `public/` directory. If that's not the case you have to extend the URI accordingly.

FLOW3 provides a standard way of resolving the URI to your MVC-Objects.

Say, you want to see the list of customers (based on the file-structure-example above). The URI to get the list would be: `http://localhost/demo/customer/list.html` or just `http://localhost/demo/customer/list`.

This URI will be resolved into the package-name (*Demo*), controller-name (*Customer*), action-name(*list*) and format-name (*html* - which is the default format).

Depending on that, the controller `\F3\Demo\Controller\CustomerController` (Pattern: `'F3\@package\Controller\@controllerController'`) and its method `listAction()` will be used. The corresponding view is `\F3\Demo\View\CustomerList` (Pattern: `'F3\@package\View\@controller@action@format'`).

If you have a look at the view pattern, you see, that you can easily add a view that creates an xml-output by creating the class `\F3\Demo\View\CustomerListXML`. You will get the xml-output by calling the URI `http://localhost/demo/customer/list.xml`.

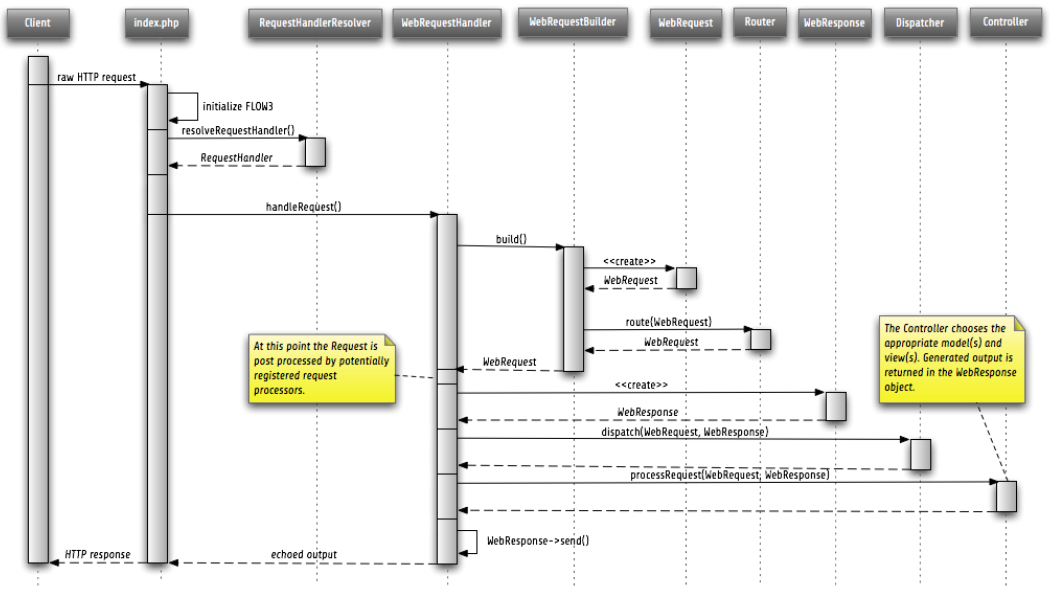
7.2. Request and Response

No matter if a FLOW3 application runs in a web context or is launched from the command line, the basic workflow is always the same: The user request is analyzed and forwarded to an appropriate controller which decides on which actions to take and finally returns a response which is handed over to the user. This section highlights the flow and the collaborators in the request-response machinery.

7.2.1. Request Processing Overview

A sequence diagram is worth a thousand words said my grandma, so let's take a look at the standard request-response workflow in FLOW3:

Figure 1.2. Example of a Web Request-Response Workflow



As you see, there are a lot of parts of the framework involved for answering a request - and the diagram doesn't even consider caching or forwarding of requests. But we didn't create this structure just for the fun of it - each object plays an important role as you'll see in the next sections.

7.2.2. Request Handler

The request handler takes the important task to handle and respond to a request. There exists exactly one request handler for each request type. By default web and command line requests are supported, but more specialized request handlers can be registered, too.

Before one of the request handlers comes to play, the framework needs to determine which of them is the most suitable for the current request. The request handler resolver asks all of the registered request handlers to rate on a scale how well they can handle the current raw request. The resolver then chooses the request handler with the most points and passes over the control.

7.2.3. Request Builder

When a request handler receives a raw request, it needs to build a request object which can be passed to the dispatcher and later to the controller. The request building delegated to a request builder which can build the required request type (ie. web, CLI etc.).

The building process mainly consists of

1. create a new request object
2. set some request-type specific parameters (like the request URI for a web request)
3. determine and set the responsible controller, action and action arguments

Especially the last step is important and requires some more or less complex routing in case of web requests.

7.2.4. Request Processors

Requests which were built by the request builder usually fit the most common needs. For special demands it is possible to postprocess the request object before it is sent to the dispatcher. Request processors can be registered through the Request Processor Chain Manager and are - as the name suggests - invoked in a chain.

7.2.5. Request Dispatcher

The final task of the MVC framework consists in dispatching the request to the controller specified in the request object. The request dispatcher will try to call the action specified in the request object and if none was specified fall back on a default action.

Note

There are more features planned for the dispatcher, but at the time of this writing they have not yet been implemented.

7.2.6. Request Types

FLOW3 supports the most important request types out of the box. Additional request types can easily be implemented by extending the `\F3\FLOW3\MVC\Request` class and registering a request handling which can handle the new request type (and takes care of building the request object). Here are the request types which come with the default FLOW3 distribution:

7.2.6.1. Web Request / Response

Web requests are the most common request types. Currently only the basic features are implemented, but further options - especially for the web response - are in the pipeline.

7.2.6.2. CLI Request / Response

Requests from the command line are recognized by the used SAPI (Server Application Programming Interface). This request type is basically the same as the generic request type and mainly exists as a marker.

7.2.6.3. AJAX Request / Response

> **Note**
>
> This request type has not yet been implemented

7.3. Controller

7.3.1. Action Controller

7.3.1.1. Initialization Methods

initializeController

initializeAction

initializeView

7.3.1.2. Configuration

7.3.1.3. Supported Request Types

7.3.1.4. Arguments

7.3.1.5. Action Methods

$this->indexActionMethodName

7.3.1.6. Action View

- $this->initializeView = TRUE | FALSE

7.3.2. Other Controllers

7.3.2.1. Abstract Controller

7.3.2.2. Request Handling Controller

7.3.2.3. Default Controller

7.4. View

7.4.1. Template View

7.4.2. Special Views

7.4.2.1. Default View

7.4.2.2. Empty View

7.5. Helpers

7.6. Model

7.7. Routing

7.8. CLI request handling

FLOW3's CLI request handling offers a comfortable and flexible way of calling code from the command line:

php index.php [*command*] [*options*] [--] [*arguments*]

command, *options* and *arguments* are optional, with varying results. The command structure follows what is commonly accpeted on unixoid systems for CLI programs:

command — If not given, the default controller of the FLOW3 package is used and it's index action is called. While this is an allowed call, it hardly makes sense (other than checking if FLOW basically works). If command is given then it is defined as *package [[sub1..N] controller action]*

First part is always the package. If only the package is given, it's default controller's index action is called.

If at least three command parts are given, the last two sepcify controller and action. Anything in between specifys a sub package structure.

Example 1.45. Some FLOW3 CLI command specifications

`testing cli run` would call the "run" action of the "cli" controller in the "Testing" package

`typo3cr admin setup default` would call the "setup" controller's "default" action in the subpackage "admin" of the package "TYPO3CR"

options — Options are either short- or long-style. The first option detected ends collecting command parts. Here are some examples:

Example 1.46. Giving options to FLOW3 CLI requests

```
-o -f=value --a-long-option --with-spaces="is possible"
--input file1 -o=file2 --event-this = works
```

arguments — Arguments can follow and will be available to the called controller in the request object. To distinguish between *command* and *arguments* in cases where no *options* are given the seperator `--` must be used.

Example 1.47. Some FLOW3 CLI commands

Calling the TYPO3CR setup:

```
php index.php typo3cr admin setup setup --dsn=sqlite:/tmp/typo3cr.db
--indexlocation=/tmp/lucene/
```

Running FLOW3 unit tests:

```
php index.php testing cli run --package-key=FLOW3 --output-
directory=./
```

Rendering the FLOW3 documentation to HTML:

```
php index.php doctools render render -p FLOW3 -o flow3-manual/
```

8. Cache Framework

9. Error and Exception Handling

10. AOP Framework

10.1. Introduction

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 behaviour 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. FLOW3'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, additional compile steps or modification of the target code – 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 [http://en.wikipedia.org]). 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.

10.1.1. AOP concepts and terminology

Let's stay with the example of a Forum for a while. The classes of the forum don't implement security themselves, but somehow we have to make sure that whenever a method `deletePost()` is called, a security check takes place. The class containing the delete method is called the *target class*. We have a new *aspect* called "security" which we'd like to *weave* into that class. Whenever the method `deletePost()` is called, a *method interceptor* defined by an *around advice* will intercept the *target method* and only proceed if the operation is allowed in the current security context.

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 FLOW3, aspects

[1] SoC could, by the way, also mean “Self-organized criticality” or “Service-oriented Computing” or refer to Google's “Summer of Code” ...

	are implemented as regular classes which are tagged by the @aspect annotation. The methods of an aspect class represent advices, the properties act as an anchor 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 FLOW3, join points are represented by the `\F3\FLOW3\AOPJoinPoint` 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 joinpoints 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. FLOW3 uses methods in an aspect class as anchors for pointcut declarations.
Pointcut expression	A poincut expression is the condition under which a joinpoint should match. It may, for example, define that joinpoints 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.

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.

10.1.2. FLOW3 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 FLOW3 (to provide a powerful, yet developer-friendly solution) when the development of TYPO3 5.0 began. We therefore decided to create a sophisticated but pragmatic implementation which adopts the concepts of AOP but takes PHP's specialities and the requirements of typical FLOW3 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.

FLOW3 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]. For the sake of simplicity and performance, pointcuts don't allow criteria which have to be evaluated at runtime (such as matching argument values of a method) and 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.

10.1.3. Implementation overview

FLOW3'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.

FLOW3 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 a 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.

10.2. Aspects

Aspects are abstract containers which accommodate pointcut-, introduction- and advice declarations. In most frameworks, including FLOW3, aspects are defined as plain classes which are tagged (annotated) as an aspect. The following example shows the definition of a hypothetical *FooSecurity* aspect:

[2] AOP was rather invented by Gregor Kiczalesand his team at the Xerox Palo Alto Research Center [http://www.parc.com/]. The original implementation was called AspectJ [http://eclipse.org/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 Group of Four and refers to the authors of the classic book *Design Patterns – Elements of Reusable Object-Oriented Software*

Example 1.48. Declaration of an aspect

```
namespace F3\MySecurityPackage;

/**
 * An aspect implementing security for Foo
 *
 * @package MySecurityPackage
 * @author  John Doe <john@typo3.org>
 * @aspect
 */
class FooSecurityAspect {

}
```

As you can see, `\F3\MySecurityPackage\FooSecurityAspect` is just a regular PHP class which may (actually must) contain methods and properties. What it makes it an aspect is solely the @aspect annotation mentioned 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.

10.3. 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 methodname, and the poincut expression, declared by an annotation. The following pointcut will match the execution of methods whose name starts with “delete”, mo matter in which class they are defined:

Example 1.49. Declaration of a named pointcut

```
   /**
  * A pointcut which matches all methods whose name starts with "delete".
  *
  *  @pointcut method(.*->delete.*())
  * @author John Doe <john@typo3.org>
  */
 public function  deleteMethods() {}
```

Declaration of the pointcut expression
Name of the pointcut

10.3.1. 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 poincut become active.

Note

The AOP framework AspectJ provides a complete pointcut language with dozens of pointcut types and expression constructs. FLOW3 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 [http://www.eclipse.org/aspectj/doc/released/progguide/index.html].

10.3.2. Pointcut designators

A pointcut expression always consists of two parts: The poincut designator and its parameter(s). The following designators are supported by FLOW3:

10.3.2.1. 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|private ClassName->methodName())
```

Specifying the visibility modifier (public, protected or private) is optional - if none is specified, any visibility will match. The class- and method name can be specified as a regular expression. Here are some examples for matching method executions:

Example 1.50. method() pointcut designator

Matches all public methods in class `\F3\MyPackage\MyObject`:

```
method(public F3\MyPackage\MyObject->.*())
```

Matches all delete methods (even protected and private ones) in any class of the package MyPackage:

```
method(F3\MyPackage\.*->delete.*())
```

Note

In other AOP frameworks, incuding 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 FLOW3 only supports matching to method execution join points anyway, we decided to simplify things by allowing only a more general `method` designator.

10.3.2.2. 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)
```

An example for the usage of this designator:

[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.

Example 1.51. class() pointcut designator

Matches all methods in class `F3\MyPackage\MyObject`:

```
class(F3\MyPackage\MyObject)
```

10.3.2.3. 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)
```

An example for the usage of `within()`:

Example 1.52. within() pointcut designator

Matches all methods in classes which implement the logger interface:

```
within(\F3\FLOW3\Log\LoggerInterface)
```

Matches all methods in classes which are part of the Foo layer:

```
within(\F3\FLOW3\FooLayerInterface)
```

10.3.2.4. classTaggedWith()

The `classTaggedWith()` designator matches on classes which are tagged with a certain annotation. As with class and method names, a regular expression can be used to describe the matching tags. The syntax of this designator is as follows:

```
classTaggedWith(tag)
```

Example 1.53. classTaggedWith() pointcut designator

Matches all classes which are tagged with an "@entity" annotation:

```
classTaggedWith(entity)
```

Matches all classes which are tagged with an annotation starting with "@cool":

```
classTaggedWith(cool.*)
```

10.3.2.5. methodTaggedWith()

The `methodTaggedWith()` designator matches on methods which are tagged with a certain annotation. As with other pointcut designators, a regular expression can be used to describe the matching tags. The syntax of this designator is as follows:

```
methodTaggedWith(tag)
```

Example 1.54. methodTaggedWith() pointcut designator

Matches all method which are tagged with an "@special" annotation:

```
methodTaggedWith(special)
```

10.3.2.6. 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. You can use this designator as follows:

Example 1.55. 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')
```

10.3.2.7. 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 `\F3\FLOW3\AOP\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 1.56. filter() pointcut designator

If the current method matches is determined by the custom filter:

```
filter(F3\MyPackage\MyCustomPointcutFilter)
```

10.3.3. 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 (ie. 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 1.57. Combining pointcut expressions

```
namespace F3\TestPackage;

/**
 * Fixture class for testing poincut definitions
 *
 * @package TestPackage
 * @aspect
 */
class PointcutTestingAspect {

 /**
  * Pointcut which includes all method executions in pointcutTestingTargetClass
  *
  * @pointcut method(F3\TestPackage\PointcutTestingTargetClass.*->.*()) && !meth
  */
 public function pointcutTestingTargetClasses() {}

 /**
  * Pointcut which consists of only the F3\TestPackage\OtherPointcutTestingTarge
  *
  * @pointcut method(F3\TestPackage\OtherPointcutTestingTargetClass->.*())
  */
 public function otherPointcutTestingTargetClass() {}

 /**
  * A combination of both above pointcuts
  *
  * @pointcut F3\TestPackage\PointcutTestingAspect->pointcutTestingTargetClasses
  * @author Robert Lemke <robert@typo3.org>
  */
 public function bothPointcuts() {}

 /**
  * A pointcut which matches all classes from the service layer
  *
  * @pointcut within(\F3\FLOW3\ServiceLayerInterface)
  */
 public function serviceLayerClasses() {}

 /**
  * A pointcut which matches any method from the BasicClass and all classes fro
  *
  * @pointcut method(F3\TestPackage\Basic.*->.*()) || within(F3\FLOW3\Service.*
  */
 public function basicClassOrServiceLayerClasses() {}
}
```

10.4. 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.

10.4.1. 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 1.58. Declaration of a before advice

```
 /**
 * Before advice which is invoked before any method call within the News packa
 *
 * @before class(F3\News\.*->.*())
 */
public function myBeforeAdvice(\F3\FLOW3\AOP\JoinPointInterface $joinPoint) {
}
```

10.4.2. After returning advice

The *after returning advice* becomes active after the target method normally returns from execution (ie. it doesn't throw an exception). After returning advices may read the result of the target method, but can't modify it.

Example 1.59. Declaration of an after returning advice

```
 /**
 * After returning advice
 *
 * @afterreturning method(public F3\News\FeedAgregator->[import|update].*()) |
 */
public function myAfterReturningAdvice(\F3\FLOW3\AOP\JoinPointInterface $joinP
}
```

10.4.3. 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 1.60. Declaration of an after throwing advice

```
 /**
 * After throwing advice
 *
 * @afterthrowing within(F3\News\ImportantLayer)
 */
public function myAfterThrowingAdvice(\F3\FLOW3\AOP\JoinPointInterface $joinPo
}
```

10.4.4. 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 1.61. Declaration of an after advice

```
 /**
 * After advice
 *
 * @after F3\MyPackage\MyAspect->justAPointcut
 */
public function myAfterAdvice(\F3\FLOW3\AOP\JoinPointInterface $joinPoint) {
}
```

10.4.5. 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 1.62. Declaration of an around advice

```
 /**
 * Around advice
 *
 * @around F3\MyPackage\MyAspect->justAPointcut
 */
public function myAroundAdvice(\F3\FLOW3\AOP\JoinPointInterface $joinPoint) {
}
```

10.5. 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 FLOW3 – you therefore can take advantage of dependency injection in case you need other objects to fullfil the task of your advice.

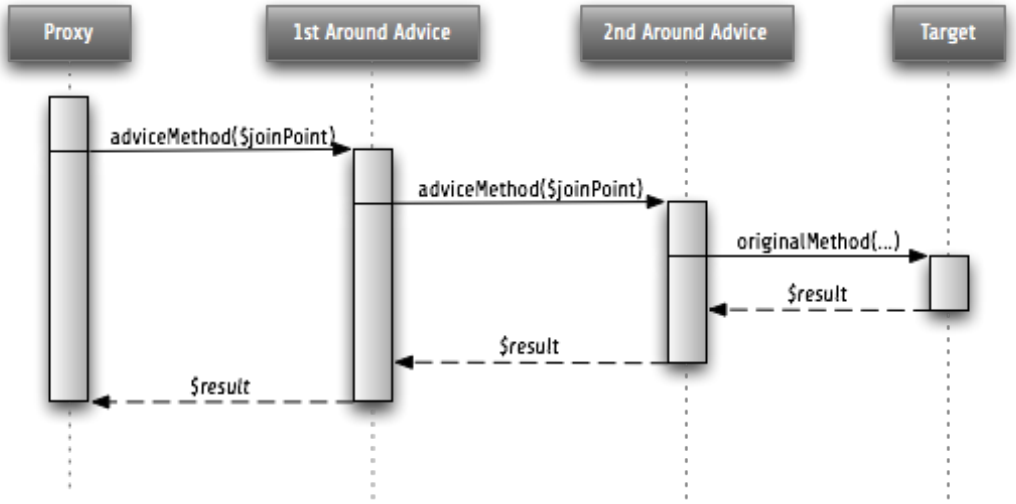
10.5.1. Accessing join points

As you have seen in the previous section, advice methods always expect an argument of the type `\F3\FLOW3\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.

10.5.2. 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.

Figure 1.3. Control flow of an advice chain



10.5.3. 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 certain package. The following code will just do that:

Example 1.63. Simple logging with aspects

```
namespace F3\MyPackage;

/**
 * A logging aspect
 *
 * @package MyPackage
 * @aspect
 */
class LoggingAspect {

 /**
  * @var \F3\FLOW3\Log\LoggerInterface A logger implementation
 protected $logger;

 /**
  * Constructor of this aspect. For logging we need a logger, which we will get
  * injected automatically by the Object Manager
  *
  * @param  \F3\FLOW3\Log\LoggerInterface $logger: An instance of a logger
  * @return void
  */
 public function __construct(\F3\FLOW3\Log\LoggerInterface $logger) {
  $this->logger = $logger;
 }

 /**
  * Before advice, logs all access to methods of our package
  *
  * @param  \F3\FLOW3\AOP\JoinPointInterface $joinPoint: The current join point
  * @return void
  * @before method(F3\MyPackage\.*->.*())
  */
 public function logMethodExecution(\F3\FLOW3\AOP\JoinPointInterface $joinPoint
  $logMessage = 'The method ' . $joinPoint->getMethodName() . ' in class ' . $j
  $this->logger->log($logMessage, 'F3\MyPackage\LoggingAspect');
 }

}
```

Note that we are using dependency injection for getting a logger instance to stay independent from any specific logging implementation. We don't have to care about the kind of logger (file based, syslog, ...) and where it comes from.

Finally an example for the implementation of an around advice: For a guestbook, 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 1.64. Implementation of an around advice

```
namespace F3\MyPackage;

/**
 * A lastname rejection aspect
 *
 * @package MyPackage
 * @aspect
 */
class LastNameRejectionAspect {

 /**
  * A pointcut which matches all guestbook submission method invocations
  *
  * @pointcut method(\F3\Guestbook\SubmissionHandlingThingy->submit())
  */
 public function guestbookSubmissionPointcut() {}

 /**
  * Around advice, rejects the lastname "Sarkosh"
  *
  * @param  \F3\FLOW3\AOP\JoinPointInterface $joinPoint: The current join point
  * @return mixed Result of the target method
  * @around F3\MyPackage\LastNameRejectionAspect->guestbookSubmissionPointcut
  */
 public function rejectLastName(\F3\FLOW3\AOP\JoinPointInterface $joinPoint) {
  if ($joinPoint->getMethodArgument('lastName') == 'Sarkosh') {
   throw new Exception('Sarkosh is not a valid lastname - 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 ...

10.6. Introductions

Introductions (also known as *Inter-type Declarations*) allow to subsequently implement an interface 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.

10.6.1. Declaring introductions

Like advices, introductions are declared by annotations. But in contrast to advices, the anchor for an introduction declaration is a property of the aspect class. The annotation tag follows this syntax:

@introduce *`NewInterfaceName`*, *`PointcutExpression`*

Although the *`PoincutExpression`* 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 1.65. Declaring introductions

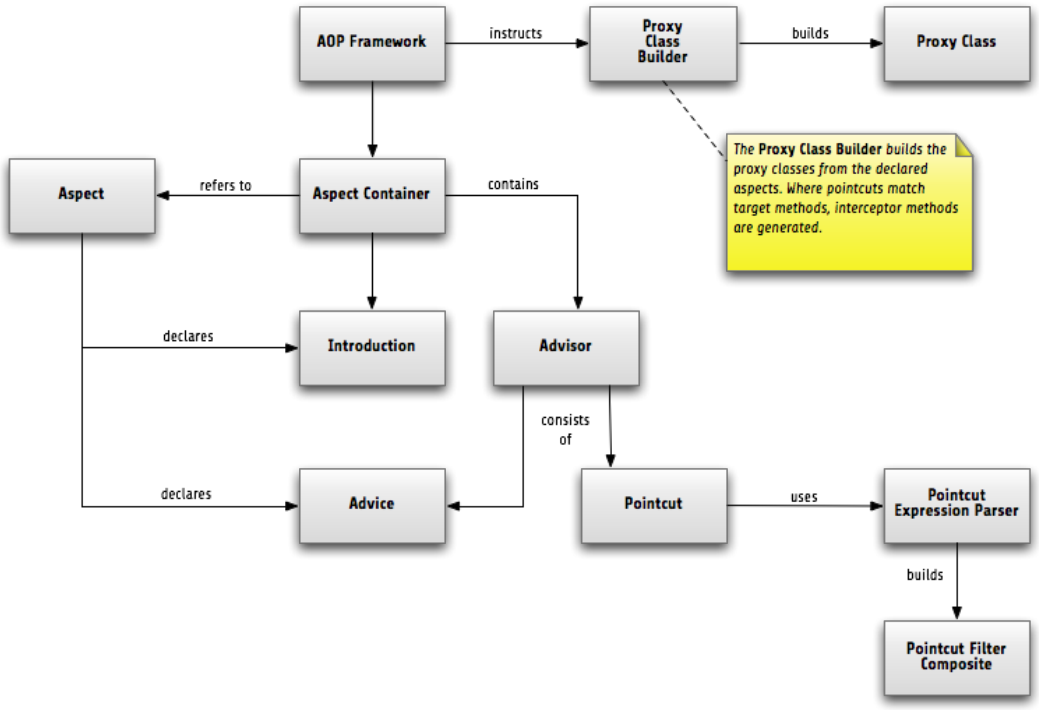
```
namespace F3\MyPackage;

/**
 * An aspect for demonstrating introductions
 *
 * @package MyPackage
 * @aspect
 */
class IntroductionAspect {

 /**
  * Introduces \F3\MyPackage\NewInterface to the class \F3\MyPackage\OldClass:
  *
  * @introduce F3\MyPackage\NewInterface, class(F3\MyPackage\OldClass)
  */
 public $newInterface;

 /**
  * Around advice, implements the new method "newMethod" of the "NewInterface"
  *
  * @param  \F3\FLOW3\AOP\JoinPointInterface $joinPoint: The current join point
  * @return void
  * @around method(F3\MyPackage\OldClass->newMethod())
  */
 public function newMethod(\F3\FLOW3\AOP\JoinPointInterface $joinPoint) {
   // We call the advice chain, in case any other advice is declared for this m
   // but we don't care about the result.
  $someResult = $joinPoint->getAdviceChain->proceed($joinPoint);

  $a = $joinPoint->getMethodArgument('a');
  $b = $joinPoint->getMethodArgument('b');
  return $a + $b;
 }
}
```

10.7. Implementation details

10.7.1. AOP proxy mechanism

The following diagram illustrates the building process of a proxy class:

Figure 1.4. Proxy building process

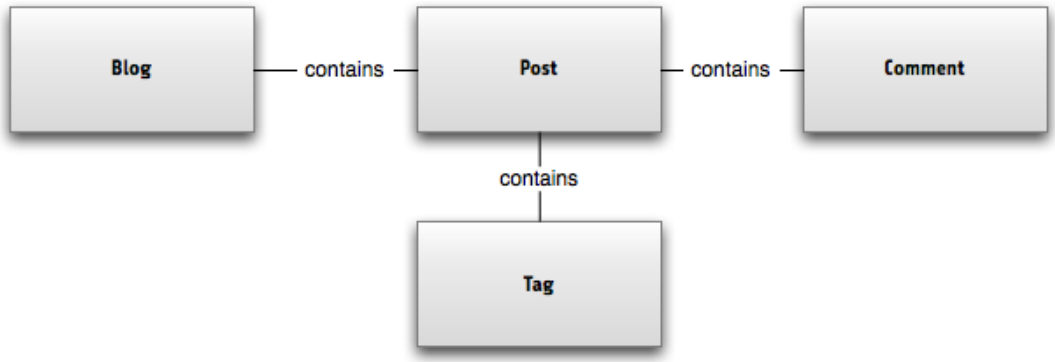


11. Persistence Framework

11.1. Introductory Example

Let's look at the following example as an introduction to how FLOW3 handles persistence. We have a domain model of a Blog, consisting of Blog, Post, Comment and Tag objects:

Figure 1.5. The objects of the Blog domain model



Connections between those objects are built by simple references in PHP, as a look at the `addPost()` method of the `Blog` class shows:

Example 1.66. The Blog's addPost() method

```
/**
 * @param \F3\Blog\Domain\Post $post
 * @return void
 */
public function addPost(\F3\Blog\Domain\Post $post) {
    $this->posts[] = $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 Blogs 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 (using FLOW3's object factory) and add it to the BlogRepository. This will automagically persist your Blog and you can retrieve it again later. No save() call needed. Oh, and the posts, commens and tags in your Blog are persisted as well, of course.

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 1.67. Persistence-related annotations in the Blog class

```
/**
 * A Blog object
 *
 * @package Blog
 * @entity
 */
class Blog {

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

    /**
     * @var array
     */
    protected $posts = array();

  ...

}
```

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?). In case of the $posts array the persistence framework persists the objects held in that array as independent objects.

Let's conclude by taking a look at the BlogRepository code:

Example 1.68. Code of a simple BlogRepository

```
/**
 * A BlogRepository
 *
 * @package Blog
 */
class BlogRepository extends \F3\FLOW3\Persistence\Repository {

    /**
     * Finds Blogs with a matching name.
     *
     * @param string $name
     * @return array
     */
    public function findByName($name) {
        $query = $this->createQuery();
        return $query->matching($query->equals('name', $name))->execute();
    }
}
```

As you can see we get away with very little code by simply extending the FLOW3-provided repository class. Nice, eh? If you like to do things the hard way you can get away with implementing `\F3\FLOW3\Persistence\RepositoryInterface` yourself.

11.2. On the priciples 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.
- 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.

11.3. Persistence-related annotations

The following table lists all annotations used by the persistence framework with their name, scope and meaning:

Table 1.2. 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.
@identifier	Variable	Marks the variable as being the object identifier. This makes the persistence backend use the value of this variable as identifier for the internal representation of the object. *You must make sure your identifier is unique, preferable use an UUID.*

11.4. Querying the storage backend

As we saw in the introductory example there is a query mechanism available that allows for relatively 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:

Figure 1.6. Object querying and reconstitution process

For the developer the complexity is hidden between the query's `execute()` method and the array of objects being returned.

Appendix A. Coding Guidelines

1. Coding Guidelines

Coding Standards are an important factor for achieving a high code quality. A common visual style, naming conventions and other technical settings allow us to produce a homogenous code which is easy to read and maintain. However, not all important factors can be covered by rules and coding standards. Equally important is the style in which certain problems are solved programmatically - it's the personality and experience of the individual developer which shines through and ultimately makes the difference between technically okay code or a well considered, mature solution.

These guidelines try to cover both, the technical standards as well as giving incentives for a common development style. These guidelines must be followed by everyone who creates code for the FLOW3 core. Because TYPO3 is based on FLOW3, it follows the same principles - therefore, whenever we mention FLOW3 in the following sections, we equally refer to TYPO3. We hope that you feel encouraged to follow these guidelines as well when creating your own packages and FLOW3 based applications.

1.1. Code formatting and layout

aka "beautiful code"

The visual style of programming code is very important. In the TYPO3 project we want many programmers to contribute, but in the same style. This will help us to:

- Easily read/understand each others code and consequently easily spot security problems or optimization opportunities
- It is a signal about consistency and cleanliness, which is a motivating factor for programmers striving for excellence

Some people may object to the visual guidelines since everyone has his own habits. You will have to overcome that in the case of FLOW3; the visual guidelines must be followed along with coding guidelines for security. We want all contributions to the project to be as similar in style and as secure as possible.

1.1.1. General considerations

- Nearly any PHP file in FLOW3 contains exactly one class and does not output anything if it is called directly. Therefore you start you file with a <?php tag and end it with the closing ?>.
- Every file must contain a header stating encoding, namespace, copyright and licensing information

  1. Declare your namespace. The namespace must start with "F3"!
  2. Because it is likely that more than one person will work on a class in the long run, we recommend adding a copyright statement like “Copyright belongs to the respective authors” and add yourself to the list of authors for the methods you implemented.
  3. If you modify a library you must document what you have changed (GPL requirement). Add your name as author / co-author to these modifications.
  4. The importance of the header is primarily to state that the code is GPL'ed. And remember only GPL compatible software is allowed to interface with FLOW3 (according to GPL itself).
  5. The copyright header itself must not start with /**, as this may confuse documentation generators!

Example A.1. The FLOW3 standard file header

```
<?php
declare(ENCODING = 'utf-8');
namespace F3\Your\Stuff\Here;

/*                                                                        *
 * This script is part of the TYPO3 project - inspiring people to share!  *
 *                                                                        *
 * TYPO3 and FLOW3 are free software; you can redistribute them and/or    *
 * modify them under the terms of the GNU General Public License version 2 *
 * as published by the Free Software Foundation.                          *
 *                                                                        *
 * This script is distributed in the hope that it will be useful, but     *
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHAN-    *
 * TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General      *
 * Public License for more details.                                       *
 *                                                                        */
```

- Code lines are of arbitrary length, no limitations to 80 characters or something similar (wake up, graphical displays have been available for decades now...)
- Lines end with a newline a.k.a `chr(10)` - UNIX style
- Files must be encoded in UTF-8

1.1.2. Indentation and line formatting

Indentation is done with tabs - and not spaces! The beginning of a line is the only place where tabs are used, in all other places use spaces. Always trim whitespace off the end of a line.

Here's a code snippet which shows the correct usage of tabs and spaces:

Example A.2. Correct use of tabs and spaces

```
/*############################################################################*
#*#Some#comment#################################################################*
#*############################################################################*/

/**
#*#Returns#the#name#of#the#currently#set#context.
#*
#*#@return#string#Name#of#the#current#context
#*#@author#Your Name <your@email.here>
#*/
public#function#getContext()#{
»   return#$this->context;
}
```

There seem to be very passionate opinions about whether TABs or spaces should be used for indentation of code blocks in the scripts. If you'd like to read more about this religious discussion, you find some nice arguments in the Joels on Software forum [http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=3978].

1.1.3. Naming

Naming is a repeatedly undervalued factor in the art of software development. Although everybody seems to agree on that nice names are a nice thing to have, most developers choose cryptic abbreviations in the end (to save some typing). Beware that we TYPO3 core developers are very passionate about naming (some people call it fanatic, well ...). In our

opinion spending 15 minutes (or more ...) just to find a good name for a method is well spent time! There are zillions of reasons for using proper names and in the end they all lead to better readable, manageable, stable and secure code.

- As a general note, english words (or abbreviations if neccessary) must be used for all class names, method names, comments, variables names, database table and field names. Although PHP6 allows for using funny japanese, tibetian or don't-know-what characters, the consensus is that english is much better to read for the most of us.
- If acronyms or abbreviations are embedded in names, keep them in the case they usually are, i.e. keep URL uppercase, also when used in a method name like `getURLForLink()`, `ACMEManager` etc.

1.1.3.1. Package names

All package names are written in UpperCamelCase. In order to avoid problems with different filesystems, only the characters a-z, A-Z and 0-9 are allowed for package names – don't use special characters.

1.1.3.2. Namespace names

Only the characters a-z, A-Z and 0-9 are allowed for namespace names – don't use special characters. All namespace names are written in `UpperCamelCase`, all uppercase names are allowed for well established abbreviations.

> **Note**
>
> When specifying class names to PHP, always reference the global namespace inside namespaced code by using a leading backslash. When referencing a class name inside a string (e.g. given to the `create`-Method of the `ObjectFactory`, in pointcut expressions or in YAML files), never use a leading backslash. This follows the native PHP notion of names in strings always being seen as fully qualified.

1.1.3.3. Class names

Only the characters a-z, A-Z and 0-9 are allowed for class names – don't use special characters.

All class names are written in `UpperCamelCase`, all uppercase names are allowed for well established abbreviations. Class names must be nouns, never adjectives.

The name of abstract classes should start with the word "Abstract".

A few examples follow:

Example A.3. Correct naming of classes

- `\F3\FLOW3\Object`
- `\F3\FLOW3\Object\Manager`
- `\F3\MyPackage\MyObject\MySubObject`
- `\F3\MyPackage\MyObject\MyHTMLParser`
- `\F3\Foo\Controller\DefaultController`
- `\F3\MyPackage\MyObject\AbstractLogger`

Example A.4. Incorrect naming of classes

- `\myClass`
- `\F3\Urlwrapper`
- `\someObject\classname`
- `\F3\MyPackage\MyObjectMySubObject`

1.1.3.4. Interface names

Only the characters a-z, A-Z and 0-9 are allowed for interface names – don't use special characters.

All interface names are written in `UpperCamelCase`, all uppercase names are allowed for well established abbreviations. Interface names must be adjectives or nouns and have the Interface suffix. A few examples follow:

Example A.5. Correct naming of interfaces

- `\F3\FLOW3\Object\ObjectInterface`
- `\F3\FLOW3\Object\ManagerInterface`
- `\F3\MyPackage\MyObject\MySubObjectInterface`
- `\F3\MyPackage\MyObject\MyHTMLParserInterface`

Example A.6. Incorrect naming of interfaces

- `\myInterface`
- `\F3\Urlwrapper`
- `\IsomeObject\classname`
- `\F3\FLOW3\Object\Manager\Interface`

1.1.3.5. Exception names

Exception naming basically follows the rules for naming classes. There are two possible types of exceptions: Generic exceptions and specific exceptions. Generic exceptions should be named "Exception" preceeded by their namespace. Specific exceptions should reside in their own sub-namespace and must not contain the word `Exception`.

Example A.7. Correct naming of exceptions

- `\F3\FLOW3\Object\Exception`
- `\F3\FLOW3\Object\Exception\InvalidClassName`
- `\F3\MyPackage\MyObject\Exception`
- `\F3\MyPackage\MyObject\Exception\OutOfCoffee`

1.1.3.6. Method names

All method names are written in lowerCamelCase, all uppercase names are allowed for well established abbreviations. In order to avoid problems with different filesystems, only the characters a-z, A-Z and 0-9 are allowed for method names – don't use special characters.

Make method names descriptive, but keep them concise at the same time. Constructors must always be called `__construct()`, never use the class name as a method name.

A few examples:

Example A.8. Correct naming of methods

- `myMethod()`
- `someNiceMethodName()`
- `betterWriteLongMethodNamesThanNamesNobodyUnderstands()`
- `singYMCALoudly()`
- `__construct()`

1.1.3.7. Variable names

Variable names are written in `lowerCamelCase` and should be

- self-explaining
- not shortened beyond recognition, but rather longer if it makes their meaning clearer

The following example shows two variables with the same meaning but different naming. You'll surely agree the longer versions are better (don't you ...?).

Example A.9. Correct naming of variables

- `$singletonObjectsRegistry`
- `$argumentsArray`
- `$aLotOfHTMLCode`

Example A.10. Incorrect naming of variables

- `$sObjRgstry`
- `$argArr`
- `$cx`

As a special exception you may use variable names like `$i`, `$j` and `$k` for numeric indexes in `for` loops if it's clear what they mean on the first sight. But even then you might want to avoid them.

1.1.3.8. Constant names

All constant names are written in UPPERCASE. This includes TRUE, FALSE and NULL. Words can be separated by underscores - you can also use the underscore to group constants thematically:

Example A.11. Correct naming of constants

- `STUFF_LEVEL`
- `COOLNESS_FACTOR`
- `PATTERN_MATCH_EMAILADDRESS`
- `PATTERN_MATCH_VALIDHTMLTAGS`

It is, by the way, a good idea to use constants for defining regular expression patterns (as seen above) instead of defining them somewhere in your code.

1.1.3.9. File names

These are the rules for naming files:

- All file names are `UpperCamelCase`.
- Class and interface files are named according to the class or interface they represent
- Each file must contain only one class or interface
- Names of files containing code for unit tests must be the same as the class which is tested, appended with "Test.php".

Here are some examples:

Example A.12. File naming in FLOW3

- `F3_TemplateEngine_TemplateEngineInterface.php`

  Contains the interface `\F3\TemplateEngine\TemplateEngineInterface` which is part of the package \F3\TemplateEngine

- `F3_Error_RuntimeException.php`

  Contains the `\F3\Error\RuntimeException` being a part of the package \F3\Error

- `F3_DataAccess_Manager.php`

  Contains class `\F3\DataAccess\Manager` which is part of the package \F3\DataAccess

- `F3_FLOW3_Package_Manager.php`

  Contains the class `\F3\FLOW3\Package\Manager` which is part of the package \F3\FLOW3

- `F3_FLOW3_Package_ManagerTest.php`

  Contains the class `\F3\FLOW3\Package\ManagerTest` which is a test case for PHPUnit.

1.1.4. PHP code formatting

1.1.4.1. Strings

In general, we use single quotes to enclose literal strings:

Example A.13. String literals

```
$vision = 'Inspiring people to share';
```

If the string itself contains single quotes or apostrophes, we use double quotes:

Example A.14. String literals enclosed by double quotes

```
$message = "'Kasper' is the name of the friendly ghost.";
```

If you'd like to insert values from variables, we recommend to concatenate strings or use double quotes in the following form:

Example A.15. Variable substitution

```
$message = 'Hey ' . $name . ', you look ' . $look . ' today!';
$message = "Hey $name, you look $look today!";
```

As you've seen in the previous example, we concatenate strings by using the dot operator. A space must be inserted before and after the dot for better readability:

Example A.16. Concatenated strings

```
$vision = 'Inspiring people ' . 'to share.';
```

You may break a string into multiple lines if you use the dot operator. You'll have to indent each following line to mark them as part of the value assignment:

Example A.17. Multi-line strings

```
$vision = 'Inspiring' .
  'people ' .
  'to ' .
  'share';
```

1.1.4.2. Arrays

1.1.4.3. Classes

...

Example A.18. Classes

```
namespace F3\MyPackage;

class MyObject {

}
```

1.1.4.4. Functions and methods

1.1.4.5. if statements

- There needs to be one space between the `if` keyword and the opening brace "("
- ...

Example A.19. if statements

```
if ($something || $somethingElse) {
  doThis();
} else {
  doSomethingElse();
}

if (allGoesWrong()) throw new Exception('Hey, all went wrong!');

if (weHaveALotOfCriteria()
  && notEverythingFitsIntoOneLine()
  || youJustTendToLikeIt()) {
    doThis();

} else {
  ...
}
```

1.2. Documentation

All code must be documented with inline comments. The syntax is that known from the Java programming language (JavaDoc). This way code documentation can automatically be generated using tools like phpDocumentor or Doxygen. The "official" tool used is phpDocumentor[1], so syntax and documentation usage are chosen to work best with it.

[1] We look into Doxygen as well, currently both tools have problems with using namespaces in PHP.

1.2.1. Documentation blocks

A file can contain different documentation blocks, relating to the file itself, the class in the file and finally the members of the class. A documentation block is always used for the entity it precedes.

1.2.1.1. File documentation

The first documentation block in the file is essential for defining the package the file and it's contents belong to. Although it would not be strictly needed to have the file level documentation block (because each file contains only one class in FLOW3), we still use it because it

- avoids warnings when rendering the documentation
- makes sure that even code outside of classes is assigned to the correct package and documented correctly

That means that each file must contain a documentation block like shown below, right below the header stating the license:

Example A.20. Standard file level documentation block

```
/**
 * @package    [packagename]
 * @subpackage [subpackage name if necessary]
 * @version    $Id$
 */
```

The package tag is mandatory, the subpackage tag is optional and should only be used if needed.

Id, Subversion and keyword expansion

The Id in the version tag will be expanded with information about the file version by Subversion. This so-called keyword expansion needs to be explicitly enabled, though! We recommend to put this into your `~.subversion/config file`:

Example A.21. Suggested configuration for Subversion in `~/.subversion/config`

```
[miscellany]
global-ignores = #*# *.rej *.orig *.bak *~ .*
log-encoding = utf-8
enable-auto-props = yes
[auto-props]
*.php = svn:keywords=Id Revision
```

This does a little more than just enable the keyword expansion, it also sets the character encoding for the log messages and makes Subversion ignore some standard backup and metadata filenames.

1.2.1.2. Class documentation

Classes have their own documentation block describing the classes purpose, assigning a package and subpackage. Very often the code within a class is expanded and modified by a number of authors. We therefore recommend to add the names of the developers to the method documentation. An exception should be the documentation for interfaces where you list all authors in the interface documentation. Exceptions itself never have an author annotation.

Example A.22. Standard class documentation block

```
/**
 * First sentence is short description. Then you can write more, just as you like
 *
 * Here may follow some detailed description about what the class is for.
 *
 * Paragraphs are seperated by a empty line.
 *
 * @package [packagename]
 * @subpackage [subpackage name if necessary]
 * @version $Id$
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License, version 2
 */
class SomeClass {
 ...
```

Example A.23. Standard interface documentation block

```
/**
 * First sentence is short description. Then you can write more, just as you like
 *
 * Here may follow some detailed description about what the interface is for.
 *
 * Paragraphs are seperated by a empty line.
 *
 * @package [packagename]
 * @subpackage [subpackage name if necessary]
 * @version $Id$
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License, version 2
 * @author Your Name <your@email.here>
 */
interface SomeInterface {
 ...
```

Example A.24. Standard exception documentation block

```
/**
 * First sentence is short description. Then you can write more, just as you like
 *
 * Here may follow some detailed description about what the interface is for.
 *
 * Paragraphs are seperated by a empty line.
 *
 * @package [packagename]
 * @subpackage [subpackage name if necessary]
 * @version    $Id$
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License, version 2
 */
class SomeException extends \Exception {
 ...
```

Additional tags or annotations, such as @see or @aspect, can be added as needed.

1.2.1.3. Documenting variables, constants, includes

Properties of a class should be documented as well. We use the short version for documenting them:

Example A.25. Standard variable documentation block

```
/**
 * A short description, very much recommended
 * @var string
 */
protected $title = 'Untitled';
```

1.2.1.4. Method documentation

For a method, at least all parameters and the return value must be documented. Please also add your name by using the @author tag. The @access tag must not be used as it makes no sense (we're using PHP >= 5 for some reason, don't we?)

Example A.26. Standard method documentation block

```
/**
 * A description for this method
 *
 * Paragraphs are seperated by a empty line.
 *
 * @param \F3\FLOW3\Log\LoggerInterface $logger A logger
 * @param string $someString This parameter should contain some string
 * @return void
 * @author Your Name <your@email.here>
 */
public function __construct(\F3\FLOW3\Log\LoggerInterface $logger, $someString) {
 ...
```

A special note about the @param tags: The parameter type and name are seperated by one space, not aligned. Do not put a colon after the parameter name. Always document the return type, even if it is void - that way it is clearly visible it hasn't just been forgotten.

1.2.2. Documentation tags

There are quite a few documentation tags that can be used. Here is a list of tags that can and should be used within the TYPO3 project:

- @author
- @copyright
- @deprecated
- @example
- @filesource
- @global
- @ignore
- @internal
- @license
- @link
- @package
- @param
- @return
- @see
- @since

- @subpackage
- @todo
- @uses
- @var
- @version

Some are useless for PHP5 and PHP6, such as the tag for declaring a variable or method private:

- @access

> **Important**
>
> If you are unsure about the meaning or ose of those tags, look them up in the phpDocumentor manual, rather than doing guesswork.

> **Note**
>
> There are more tags which are used in FLOW3, however their purpose is not documention but configuration. Currently annotations are also used for defining aspects and advices for the AOP framework as well as for giving instructions to the Persistence framework.

1.3. Coding

1.3.1. Overview

This section gives you an overview of FLOW3's coding rules and best practices.

1.3.2. General PHP best practices

- All code should be object oriented. This means there should be no functions outside classes if not absolutely necessary. If you need a "container" for some helper methods, consider creating a static class.
- All code must make use of PHP5 / PHP6 advanced features for object oriented programming.
  - Use PHP namespaces (see http://www.php.net/manual/language.namespaces.php)
  - Always declare the scope (public, protected, private) of classes and member variables
  - Make use of iterators and exceptions, have a look at the SPL (see http://www.php.net/manual/ref.spl.php)
  - Make use of type-hinting wherever possible (see http://www.php.net/manual/language.oop5.typehinting.php)
- Always use <?php as opening tags (never only <?)
- Add an encoding declaration as the first line of your PHP code, followed by the namespace declaration. For TYPO3 the encoding must be UTF-8

Example A.27. Encoding statement for .php files

```
<?php
declare(ENCODING = 'utf-8');
namespace F3\Your\Stuff\Here;

...
```

1.3.2.1. Comments

In general, comments are a good thing and we strive for creating a well-documented source code. However, inline comments can often be a sign for a bad code structure or method naming.[2]As an example, consider the following code:

Example A.28. Bad coding smell: Comments

```
  // We only allow valid persons:
if (is_object($p) && strlen($p->lastN) > 0 && $p->hidden === FALSE && $this->environment->moonPhase =
  $xmM = $thd;
}
```

This is a perfect case for the refactoring technique "extract method": In order to avoid the comment, create a new method which is as explanatory as the comment:

Example A.29. Smells better!

```
if ($this->isValidPerson($person) {
  $xmM = $thd;
}
```

Bottom line is: You may (and are encouraged to) use inline comments if they support the readability of your code. But always be aware of possible design flaws you probably try to hide with them.

1.3.3. Error handling and exceptions

FLOW3 makes use of a hierarchy for its exception classes. The general rule is to throw preferably specific exceptions and usually let them bubble up until a place where more general exceptions are catched. Consider the following example:

Some method tried to retrieve an object from the object manager. However, instead of providing a string containing the object name, the method passed an object (of course not on purpose - something went wrong). The object manager now throws an `InvalidObjectName` exception. In order to catch this exception you can, of course, catch it specifically - or only consider a more general `Object` exception (or an even more general `FLOW3` exception). This all works because we have the following hierarchy:

```
+ \F3\FLOW3\Exception
  + \F3\FLOW3\Object\Exception
    + \F3\FLOW3\Object\Exception\InvalidObjectName
```

1.3.4. Cross platform issues

[2]This is also referred to as a bad "smell" in the theory of Refactoring. We highly recommend reading "Refactoring" by Martin Fowler - if you didn't already.