AOP PHP EXTENSION

JULIEN SALLEYRON, GÉRALD CROËS

This Page Intentionally Left Blank

Table of contents

Chapter 1

Introduction

AOP is a PECL extension that enables you to use Aspect Oriented Programming in PHP, without the need to compile or proceed to any other intermediate step before publishing your code.

The AOP extension is designed to be the easiest way you can think of for integrating AOP to PHP.

AOP aims to allow separation of cross-cutting concerns (cache, log, security, transactions, ...)

1.1 PHP's AOP extension history

The AOP extension is a project which started a while ago even if its development is quite very new (early 2012). It was first expected to be a fully PHP developed library, as part of a dependency injection framework. The Aspect Oriented Programming implementation would have taken the form of auto generated proxies.

That was before Julien Salleyron, the lead developer of the project, wanted to take it to the next level while writing the AOP core features as a PHP's extension.

Gérald Croës also belongs to the initial team, mainly in charge of the documentation and discussions around the extension's API.

1.2 Installation

Download the AOP from github, compile and add the extension to your php.ini

```
#Clone the repository on your computer
git clone https://github.com/AOP-PHP/AOP
```

```
cd AOP
#prepare the package, you will need to have development tools for php
phpize
#compile the package
./configure
make
#before the installation, check that it works properly
make test
#install
make install
```

Now you can add the following line to your php.ini to enables AOP

```
extension=aop.so
```

1.3 What is AOP ? Basic tutorial

Let's assume the following class

```
class MyServices
{
   public function doAdminStuff1 ()
   {
      //some stuff only the admin should do
      echo "Calling doAdminStuff1";
   }

   public function doAdminStuff2 ()
   {
      //some stuff only the admin should do
      echo "Calling doAdminStuff2";
   }
}
```

Now you want your code to be safe, you don't want non admin users to be able to call doAdminMethods.

What are your solutions ?

- Add some code to check the credentials "IN" you MyServices class. The drawback is that it will pollute your code, and your core service will be less readable.
- Let the clients have the responsibility to check the credentials when required. The drawbacks are that you will duplicate lots of code client side if you have to call the service from multiple places

- Add some kind of credential proxy that will check the credentials before calling the actual service. The drawbacks are that you will have to write some extra code, adding another class on the top of your services.

Moreover, these solutions tend to increase in complexity while you are adding more cross-cutting concerns like caching or logging.

That's where AOP comes into action as you will be able to tell PHP to do some extra actions while calling your MyServices's admin methods.

So let's first write the rule needed to check if we can or cannot access the admin services.

```
function adviceForDoAdmin ()
{
   if ((! isset($_SESSION['user_type'])) || ($_SESSION['user_type'] !==
'admin')) {
      throw new Exception('Sorry, you should be an admin to do this');
   }
}
```

Dead simple : we check the current PHP session to see if there is something telling us the current user is an admin (of course we do realize that you may have more complex routines to do that, but we'll keep this for the example).

Now, let's use AOP to tell PHP to execute this method "before" any execution of admin methods.

```
aop_add_before('MyServices->doAdmin*()', 'adviceForDoAdmin');
```

Now, each time you'll invoke a method of an object of the class MyServices, starting with doAdmin, AOP will launch the function basicAdminChecker *before* the called method.

That's it. Simple ain't it ?

Now le's try the examples :

```
//session is started and we added the above examples to configure
MyServices & basicAdminChecker

$services = new MyServices();
try {
   $services->doAdminStuff1();//will raise an exception as nothing in
the current session tells us we are an admin
} catch (Exception $e) {
```

```
    echo "You cannot access the service, you're not an admin";
}

$_SESSION['user_type'] = 'admin';//again, this is ugly for the sake of
the example

try {
    $service->doAdminStuff1();
    $service->doAdminStuff2();
} catch (Exception $e) {
    //nothing will be caught here, we are an admin
}
```

Here you are, you know the basics of AOP.

1.4 AOP Vocabulary and PHP's AOP capabilities

1.4.1 Advice

An advice is a piece of code that can be executed. In our first example the function adviceForAdmin is an advice, it *could* be executed.

In PHP's AOP extension an advice can be a trait, a callback, an anonymous function, a static method of a class, a method of a given object or a closure.

1.4.2 Join points

Join points are places where we can attach advices.

In PHP's AOP extension, a join point can be:

- before any method / function call
- after any method / function call
- around any method / function call
- During the arousing of an exception of any method / function
- after any method / function call, should the method terminate normally or not (triggers an exception or not)

In our first example, we used a "before" join point.

1.4.3 Pointcut

Pointcuts are a way to describe whether or not a given join point will trigger the execution of an advice.

In PHP's AOP extension pointcuts can be configured with a quite simple and straightforward syntax.

In our first example the pointcut was "MyServices->doAdmin*()" and was configured to launch the advice "before" the execution of the matching methods join points.

1.5 Why or should I use AOP ?

AOP is a whole different way of thinking for developing application. It is as different as object oriented programming can be opposed to procedural programming.

Even if you don't want to base your future development on this approach, you may find it very useful for debugging purposes. Imagine a world where you can debug or get information on your code based only on information collected for a given user, a given context, a given procedure. A world where you can hunt weird and old code execution without even trying to update multiple and sparse PHP files, but just by adding advices on given conditions.

We are sure that this extension will soon be part of your future development workflow !

This Page Intentionally Left Blank

Chapter 2

PHP's AOP in action

2.1 aop_add_before

Before kind of advice enables you to

- launch advice before the execution of a given function, without interrupting anything
- launch advice before the execution of a given function, and to interrupt its execution while raising an exception
- launch advice before the execution of a given function, and to update the targeted function's arguments
- launch advice before reading and / or writing an object's property

2.1.1 A simple advice execution

```
class MyServices
{
   public function doStuff ()
   {
      echo "do my best stuff !";
   }
}

//creating the advice as a closure
$advice = function () {
    echo "I was called before doing stuff...";
};
```

```
aop_add_before('MyServices->doStuff()', $advice);

$services = new MyServices();
$services->doStuff();
```

will output

```
I was called before doing stuff...do my best stuff !
```

2.1.2 An advice that can interrupt the execution of a function / method

```
class MyServices
{
   public function doStuff ()
   {
      echo "do my best stuff !";
   }
}

//the advice is a simple function
function adviceToInterruptDoStuff ()
{
   if (! isset($_SESSION['user'])) {
      throw new Exception ("I will never do that with someone I don't
know");
   }
}

aop_add_before('MyServices->doStuff()', 'adviceToInterruptDoStuff');

$services = new MyServices();
try {
    $services->doStuff();
} catch (Exception $e) {
    echo $e->getMessage();
}
```

will output

```
I will never do that with someone I don't know
```

2.1.3 An advice that can update the advice function's arguments

```
class MyServices
{
```

```
    public function doStuff ($name)
    {
        echo "I'll do my best stuff for $name !";
    }
}

//the advice is a simple function
function adviceToUpdateArguments (AopJoinPoint $object)
{
    $args = $object->getArguments();
    if ($args[0] === null) {
        $args[0] = 'anyone';
        $object->setArguments($args);
    }
}

aop_add_before('MyServices->doStuff()', 'adviceToUpdateArguments');

$services = new MyServices();
$services->doStuff(null);
```

will output

```
I'll do my best stuff for anyone !
```

2.1.4 An advice that knows you're dealing with an object's property

```
class Paparazzi
{
    public function alert ()
    {
        echo "Celebrity will act or say something !";
    }
}

class Celebrity
{
    public $publicStuff = 'public thinking';
    private $secretStuff;
    public function act ()
    {
        $this->secretStuff = 'secret';
    }
```

```
    public function say ()
    {
        echo $this->publicStuff;
    }
}

//the advice is a simple function
$paparazzi = new Paparazzi();
$paparazzi_informer = function () use ($paparazzi)
{
    $paparazzi->alert();
};
//As no parenthesis are given in the selector, the advice deals with
properties (read / write)
aop_add_before('Celebrity->*Stuff', $paparazzi_informer);

$CynthiaBellulla = new Celebrity();
$CynthiaBellulla->act();
echo $CynthiaBellulla->say();
```

will output

```
Celebrity will act or say something !Celebrity will act or say something
!public thinking
```

2.1.5 An advice that is interested only in properties writing

```
class Paparazzi
{
    public function alert ()
    {
        echo "Celebrity secretly act on something !";
    }
}

class Celebrity
{
    private $secretStuff;
    public function act ()
    {
        //Reads the property (won't trigger the pointcut as it's a read
operation)
        $oldValue = $this->secretStuff;

        //Writing the new value.
        $this->secretStuff = 'secret';
```

```
    }
}

//the advice is a simple function
$paparazzi = new Paparazzi();
$paparazzi_informer = function () use ($paparazzi)
{
    $paparazzi->alert();
};
//will be triggered before writing the property privateStuff
aop_add_before('write Celebrity->secretStuff', $paparazzi_informer);

$CynthiaBellulla = new Celebrity();
$CynthiaBellulla->act();
```

will output

```
Celebrity secretly act on something !
```

2.1.6 An advice that knows what happened on the hooked properties

If you want to, you can accept in your advice an AopJoinPoint object that will gives your advice more information on what exactly happened.

```
class Paparazzi
{
    public function shoot ($who, $what, $from)
    {
        echo "I'm taking pictures of $who doing some $what (that's
supposed to be $from)";
    }
}

class Celebrity
{
    private $secret;

    private $name;

    public function __construct ($name)
    {
        $this->name = $name;
    }
```

```
    public function getName ()
    {
        return $this->name;
    }

    public function act ()
    {
        //Reads the property (won't trigger the pointcut as it's a read
operation)
        $oldValue = $this->secret;

        //Writing the new value.
        $this->secret = 'shoping at london';
    }
}

//the advice is a simple function
$paparazzi = new Paparazzi();
$paparazzi_informer = function (AopJoinPoint $aop_tjp) use ($paparazzi)
{
    if ($aop_tjp->getKindOfAdvice() === AOP_KIND_BEFORE_READ_PROPERTY) {
        return;//we don't care if the value is just readed
    } elseif ($aop_tjp->getKindOfAdvice() ===
AOP_KIND_BEFORE_WRITE_PROPERTY) {
        $paparazzi->shoot($aop_tjp->getObject()->getName(),//calls getName
on the caught celebrity
                          $aop_tjp->getAssignedValue(),//gets the value
that should be assigned to $object->secret
                          $aop_tjp->getPropertyName()//the name of the
caught property
                          );
    }
};

//will be triggered before writing the property privateStuff
aop_add_before('Celebrity->secret', $paparazzi_informer);

$CynthiaBellulla = new Celebrity('Cynthia Bellula');
$CynthiaBellulla->act();
```

will output

```
I'm taking pictures of Cynthia Bellula doing some shoping at london
(that's supposed to be secret)
```

2.1.7 An advice that updates the assigned value of a property

```
class Developper
{
   public $preferences;
}

$spread_the_love = function (AopJoinPoint $aop_tjp)
{
   $assigned = $aop_tjp->getAssignedValue();
   if ($assigned !== 'PHP') {
       $assigned .= ' and PHP';
   }
};

//will be triggered before writing the property privateStuff
aop_add_before('write Developper->preferences', $spread_the_love);

$developper = new Developper();
$developper->preferences = 'Java';

echo "This developper loves ", $developper->preferences;
```

will output

```
This developper loves Java and PHP
```

2.2 aop_add_after

After kind of advice enables you to

- do stuff after the matched joinpoint
- replace the return of the adviced function
- launch an exception in case of an incorrect / unwanted return

2.2.1 doing stuff after the triggered joinpoint

```
class MyServices
{
   public function doStuff ()
   {
      return "do my best stuff !";
   }
}
```

```
//creating the advice as a closure
$advice = function () {
    echo "Did some stuff !\n";
};

aop_add_after('MyServices->doStuff()', $advice);

$services = new MyServices();
echo $services->doStuff();
```

will output

```
Did some stuff !
do my best stuff !
```

Here you can see that the advice is called right after the execution of the triggered joinpoint (return "do my best stuff !"), but before anything else can occur (echo $services->doSuff()).

2.2.2 replacing / updating the return value of a triggered joinpoint

```
class MyServices
{
   public function doStuff ()
   {
      return "doing my best stuff !";
   }
}

//creating the advice as a closure
$advice = function (AopJoinPoint $joinpoint) {
    $returnValue = $joinpoint->getReturnedValue();
    $returnValue = str_replace('best', 'very best', $returnValue);
    $joinpoint->setReturnedValue($returnValue);
};

aop_add_after('MyServices->doStuff()', $advice);

$services = new MyServices();
echo $services->doStuff();
```

will output

```
doing my very best stuff !
```

2.2.3 launching an exception in case of unwanted / incorrect returned value

Here we will ask PHP's AOP extension to raise an exception if a call to file_get_contents returns FALSE (error). This may not be a best practice as it could add overhead to native PHP functions, but such a practice can be useful if you're using an old PHP library that is not using exceptions as a mean to raise errors.

```
//creating the advice as a closure
$advice = function (AopJoinPoint $joinpoint) {
    $args = $joinpoint->getArguments();
    if ($joinpoint->getReturnedValue() === false) {
       throw new Exception("Cannot read from file '{$args[0]}'");
    }
};

aop_add_after('file_get_contents()', $advice);

try {
   @file_get_contents('foo file that does not exists');
} catch (Exception $e) {
   echo $e->getMessage();
}
```

will output

```
Cannot read from file 'foo file that does not exists'
```

2.3 aop_add_around

Around kind of advice enables you to

- completely replace the matched joinpoint (including raising exceptions)
- do stuff around (before and / or) after the joinpoint, including catching exceptions
- replacing arguments of the matching joinpoint (as of the before kind of advice)
- replacing the return of the matching joinpoint (as of the after kind of advice)
- and of course a mix of all of the above

2.3.1 Replace the matching join point

2.3.1.1 Without any consideration of the triggered joinpoint

```
class MyGoodServices
{
   public function doStuff ($name)
```

```
    {
        echo "I'll do my best stuff for $name !";
    }
}

//the advice is a static function of a given class
class Evil
{
    public static function advice (AopJoinPoint $object)
    {
        echo "I'll do the worst stuff I can to everyone ... mouhahahahaha
!";
    }
}

aop_add_around('MyGoodServices->doStuff()', array('evil', 'advice'));

$services = new MyGoodServices();
$services->doStuff('you');
```

will output

```
I'll do the worst stuff I can to everyone ... mouhahahahaha !
```

2.3.1.2 Taking into account some considerations of the matched joinpoint

```
class MyGoodServices
{
    public function doStuff ($name)
    {
        echo "I'll do my best stuff for $name !";
    }
}

//the advice is a simple method of an object
class Evil
{
    function advice (AopJoinPoint $object)
    {
        $args = $object->getArguments();
        echo "I'll do the worst stuff I can to {$args[0]} ! ...
mouhahahahaha !";
    }
}
```

```
$evil = new Evil();
aop_add_around('MyGoodServices->doStuff()', array($evil, 'advice'));

$services = new MyGoodServices();
$services->doStuff('you');
```

will output

```
I'll do the worst stuff I can to you ! ... mouhahahahaha !
```

2.3.2 Around the triggered joinpoint

```
class DivideByZeroException extends Exception {}

class DivideServices
{
   public function divide ($number, $divideBy)
   {
      if ($divideBy == 0) {
         throw new DivideByZeroException("Cannot divide by zero");
      }
      echo $number / $divideBy;
   }
}

//the advice is a static function of a given class
$advice = function (AopJoinPoint $joinpoint) {
    //do stuff before
    $args = $joinpoint->getArguments();
    echo " {$args[0]} by {$args[1]} equals [";
    try {
        echo $joinpoint->process();//asks for the joinpoint to be
processed as normal
    } catch (DivideByZeroException $e) {
       echo "Infinity";
    }
    echo "]";//do stuff after
};

aop_add_around('DivideServices->divide()', $advice);

$services = new DivideServices();
$services->divide(4, 2);
$services->divide(4, 0);
```

will output

```
 4 by 2 equals [2] 4 by 0 equals [Infinity]
```

2.3.3 An advice that can update the advice function's arguments

```
class MyServices
{
   public function doStuff ($name)
   {
      echo "I'll do my best stuff for $name !";
   }
}

//the advice is a simple function
function adviceUpdatingArguments (AopJoinPoint $object)
{
   $args = $object->getArguments();
   if ($args[0] === null) {
      $args[0] = 'anyone';
      $object->setArguments($args);
   }
   $object->process();
}

aop_add_around('MyServices->doStuff()', 'adviceUpdatingArguments');

$services = new MyServices();
$services->doStuff(null);
```

will output

```
I'll do my best stuff for anyone !
```

2.3.4 replacing / updating the return value of a triggered joinpoint

```
class MyServices
{
   public function doStuff ()
   {
      return "doing my best stuff !";
   }
}

//creating the advice as a closure
```

```php
$advice = function (AopJoinPoint $joinpoint) {
    $joinpoint->process();
    $returnValue = $joinpoint->getReturnedValue();
    $returnValue = str_replace('best', 'very best', $returnValue);
    $joinpoint->setReturnedValue($returnValue);
};

aop_add_around('MyServices->doStuff()', $advice);

$services = new MyServices();
echo $services->doStuff();
```

will output

```
doing my very best stuff !
```

2.4 AopJoinPoint complete reference

An instance of AopJoinPoint will always be passed to your advice. This object contains several information, such as the pointcut which triggered the joinpoint, the arguments, the returned value (if available), the raised exception (if available), and will enable you to run the expected method in case you are "around" it.

2.4.1 getKindOfAdvice

This will tell in which condition your advice was launched

- AOP_KIND_BEFORE before a given call, may it be function, method or property access (read / write)
- AOP_KIND_BEFORE_METHOD before a method call (method of an object)
- AOP_KIND_BEFORE_FUNCTION before a function call (not a method call)
- AOP_KIND_BEFORE_PROPERTY before a property (read or write)
- AOP_KIND_BEFORE_READ_PROPERTY before a property access (read only)
- AOP_KIND_BEFORE_WRITE_PROPERTY before a property write (write only)
- AOP_KIND_AROUND around a given call, may it be function, method or property access (read / write)
- AOP_KIND_AROUND_METHOD around a method call (method of an object)
- AOP_KIND_AROUND_FUNCTION around a function call (not a method call)
- AOP_KIND_AROUND_PROPERTY around a property (read or write)
- AOP_KIND_AROUND_READ_PROPERTY around a property access (read only)

- AOP_KIND_AROUND_WRITE_PROPERTY around a property write (write only)
- AOP_KIND_AFTER after a given call, may it be function, method or property access (read / write)
- AOP_KIND_AFTER_METHOD after a method call (method of an object)
- AOP_KIND_AFTER_FUNCTION after a function call (not a method call)
- AOP_KIND_AFTER_PROPERTY after a property (read or write)
- AOP_KIND_AFTER_READ_PROPERTY after a property access (read only)
- AOP_KIND_AFTER_WRITE_PROPERTY after a property write (write only)

2.4.2 getArguments

getArguments will return the triggering method arguments as an indexed array. The resulting array will give values when the triggering method expected values, and references where the triggering method expected references.

```
function callMe ($name, & $reference)
{
   echo "$name and $reference. ";
}

$advice = function (AopJoinPoint $joinpoint) {
   $args = $joinpoint->getArguments();
   $args[0] = 'NEW Name';//won't update the original $name parameter as
it is a value
   $args[1] = 'UPDATED $reference';//WILL update the original $reference
parameter as it is a reference
};

aop_add_before('callMe()', $advice);

$name = "name";
$reference = "reference";
callMe($name, $reference);
echo "After the method execution, value of name is $name and value of
reference is $reference";
```

will output

```
name and UPDATED $reference. After the method execution, value of name
is name and value of reference is UPDATED $reference
```

2.4.3 setArguments

setArguments enables you to replace all the arguments the triggering method will receive. Beware that if you want to keep references you will have to explicitly pass them back to setArguments.

```
function callMe ($name, & $reference, & $reference2)
{
   echo "$name, $reference and $reference2. ";
   $name = "M - $name";
   $reference = "M - $reference";
   $reference2 = "M - $reference2";
}

$advice = function (AopJoinPoint $joinpoint) {
   $args = $joinpoint->getArguments();

   $args[0] = "NEW {$args[0]}";
   $args[1] = "NEW {$args[1]}";
   $args[2] = "NEW {$args[2]}";

   $newArgs = array();
   $newArgs[0] = $args[0];
   $newArgs[1] = & $args[1];//the reference is kept
   $newArgs[2] = $args[2];//newArgs carry a copy of $args[2], the advice
won't be able to update it's value

   $joinpoint->setArguments($newArgs);
};

aop_add_before('callMe()', $advice);

$name = "name";
$reference = "reference";
$reference2 = "reference2";
callMe($name, $reference, $reference2);
echo "After the method execution, value of name is $name, reference is
$reference and reference2 is $reference2";
```

will output

```
NEW name, NEW reference and NEW reference2. After the method execution,
value of name is name, reference is M - NEW reference and reference2 is
NEW reference2
```

As a rule of thumb, if you don't want to mind about references, keep the arguments in the resulting array to update their values and give the array back back to setArguments.

```
function callMe ($name, & $reference, & $reference2)
{
   echo "$name, $reference and $reference2. ";
   $name = "M - $name";
   $reference = "M - $reference";
   $reference2 = "M - $reference2";
}

$advice = function (AopJoinPoint $joinpoint) {
   $args = $joinpoint->getArguments();
   $args[0] = "NEW {$args[0]}";
   $args[1] = "NEW {$args[1]}";
   $args[2] = "NEW {$args[2]}";
   $joinpoint->setArguments($args);
};

aop_add_before('callMe()', $advice);

$name = "name";
$reference = "reference";
$reference2 = "reference2";
callMe($name, $reference, $reference2);
echo "After the method execution, value of name is $name, reference is
$reference and reference2 is $reference2";
```

will output

```
NEW name, NEW reference and NEW reference2. After the method execution,
value of name is name, reference is M - NEW reference and reference2 is
M - NEW reference2
```

NOTE : you should only use setArguments while processing advice of kind *before* and *around*, otherwise it might be confusing to update values of the arguments *after* the execution of the trigering method.

2.4.4 getReturnedValue

getReturnedValue will give you the returned value of the triggering method. getReturnedValue will only be populated in advice of the kind "after". In every other kind of advice getReturnedValue will be null.

If the triggering method returns a reference and you want to update the given reference you will have to explicitly ask for the reference while calling getReturnedValue.

```
class Writer
{
   protected $text;

   public function & getText ()
   {
      $this->text = "some text";
      return $this->text;
   }

   public function echoText ()
   {
      echo $this->text;
   }
}

$advice = function (AopJoinPoint $joinpoint) {
   //You're asking explicitely for the reference
   $result = & $joinpoint->getReturnedValue();
   //Updating the value of the reference
   $result = "This is the new text";
};

aop_add_after("Writer->getText()", $advice);

$writer = new Writer();
$text = $writer->getText();
$writer->echoText();
```

will output

```
"This is the new text"
```

If you do the same without the use of reference the value of "Writer->foo" won't be updated, e.g.:

```
class Writer
{
   protected $text;

   public function & getText ()
   {
```

```
        $this->text = "some text";
        return $this->text;
    }

    public function echoText ()
    {
        echo $this->text;
    }
}

$advice = function (AopJoinPoint $joinpoint) {
    //You're NOT asking explicitely for the reference
    $result = $joinpoint->getReturnedValue();
    //The returned value of the trigerring method won't be updated
    $result = "This is the new text";
};

aop_add_after("Writer->getText()", $advice);

$writer = new Writer();
$text = $writer->getText();
$writer->echoText();
```

will output

```
some text
```

NOTE : Of course if the triggering method doesn't return a reference asking or not for the reference won't make any difference.

2.4.5 setReturnedValue

setReturnedValue enables you to define the resulting value of the triggering method. This function makes sense for advice of kind after, around, exception and final.

If you are assigning a returned value to a method that was expected to return a reference the original reference will be lost and won't be replaced. To replace the content of an original reference just proceed as explained in the getReturnedValue documentation.

2.4.6 process

The process method allows you to explicitly launch the triggering method or property operation (read / write).

The process method will only be available for advice of kind around. Any call to process in advice of other kinds will launch an AopException with a message like "Cannot launch the process method in an advice of kind XXX".

2.4.7 getPointcut

getPointcut returns the pointcut (as a string) that triggered the joinpoint.

2.4.8 getObject

getObject returns the object of the triggered joinppoint. If the joinpoint does not belongs to an object, getObject returns null.

2.4.9 getClassName

getClassName returns the object's class name of the triggered joinpoint. If the joinpoint does not belongs to a class, getClassName returns null.

If the class is declared in a namespace getClassName indicates the full name of the class (with the namespace).

2.4.10 getMethodName

getMethodName returns the name of the method of the triggered joinpoint. If the joinpoint was triggered by a property operation it will raise an error. If the joinpoint was triggered by a function operation it will raise an error.

2.4.11 getFunctionName

getFunctionName returns the name of the function of the triggered joinpoint. If the joinpoint was triggered by a property operation it will raise an error. If the joinpoint was triggered by a method operation it will raise an error.

2.4.12 getPropertyName

getPropertyName returns the name of the property of the triggered joinpoint. If the joinpoint was triggered by a method operation it will raise an error.

2.4.13 getAssignedValue

getAssignedValue returns the value assigned to the property of the triggered joinpoint. If the joinpoint was triggered by a method operation it will raise an error. If the joinpoint was triggered by a read operation it will also raise an error.

2.5 Pointcuts syntax

2.5.1 Basics

Selectors will enables you to describe with a very simple syntax functions, methods and properties that should be considered for raising the execution of a given advice.

At their simpliest form selectors will be given the name of the function itself including its namespace, followed by parenthesis.

eg :

- functionName() will raise advice for every call of the function functionName
- namespaceOne\namespaceTwo\functionName() will raise advice for every call of the function functionName in the namespace namespaceOne\namespaceTwo, but won't be triggered if you're calling only a method named functionName in an another namespace.

Of course you can specify a method of an object of a given class name by separating the class name and the method name by "->".

eg :

- MyClass->myMethod() will be triggered while calling the method myMethod of any instance of the class MyClass
- \namespaceOne\namespaceTwo\MyClass->myMethod() will be triggered while calling the method myMethod of any instance of the class MyClass in the namespace \namespaceOne\namespaceTwo.

If you want to work on properties the syntax for methods is to be considered, you simply have to omit the final parenthesis.

eg :

- MyClass->myProperty will be triggered while using the property myProperty of any instance of the class MyClass
- \namespaceOne\namespaceTwo\MyClass->myProperty will be triggered while using the property myProperty of any instance of the class MyClass in the namespace \namespaceOne\namespaceTwo.

2.5.2 public / protected / private

There is a specific keyword you can use to tell AOP to consider only methods / properties that are public, protected or private.

eg :

- public MyClass->myMethod() will be triggered while calling public methods named myMethod.
- public|protected MyClass->myMethod() will be triggered while calling public or protected methods named myMethod.
- public MyClass->myProperty will be triggered while using a public property named myProperty on an object of class MyClass.
- public|protected MyClass->myProperty will be triggered while using a public or protected property named myProperty of an object of class MyClass.

For those keywords, you can use a negation with the exclamation mark (!)

eg :

- !public MyClass->*() will accept every non public method call of objects of type MyClass in the root namespace
- !public MyClass->* will accept every non public property operation on objects of type MyClass in the root namespace

2.5.3 Wildcards

Of course you may not want to list for the AOP extension every functions of every class you're interested in having pointcuts for. There are cases where you would prefer to tell AOP the format of those elements, and that's why there are wildcards.

- *() will accept any function call in the root namespace
- admin*() will accept any function call in the root namespace which name starts with admin
- *admin() will accept any function call in the root namespace which name ends with admin
- namespaceOne/*() will accept any function call in the namespace namespaceOne, but not in subnamespaces of namespaceOne
- namespaceOne/namespaceTwo/*() will accept any function call in the namespace namespaceOne/namespaceTwo only

- /() will accept any function call in any single level namespace (eg namespaceOne/functionName or namespaceTwo/otherFunctionName but not namespaceOne/namespaceTwo/functionName)
- admin//cache*() will accept functions called with names names that starts with cache in a namespace called admin something with a second level namespace of any name (eg adminStuff/anything/cacheStuff)

Wildcards can also be used to specify class names.

- *::methodName() will accept all methods call named methodName in any object (eg Object1::methodName, Object2::methodName in the root namespace)
- *Foo*::admin*() will accept methods call that start with admin in classes that contains Foo in their names, in the root namespace

2.5.4 Wildcards

- '*' match anything inside a name but stops when it encounters a /
- '**' match anything, the scope includes the paths (/)

2.5.5 Simple selectors examples

2.5.5.1 For functions

End the selector with parenthesis ()

- 'functionName()' represent any call of a function called 'functionName' in the root namespace
- 'namespaceName\functionName()' represent any call of a function called 'functionName' in the namespaceName namespace

2.5.5.2 For methods

Start your selector with a Classname (Interface or Traits will also work, inheritance is taken into account)

- 'ClassName->methodName()' represent any call of a method called methodName from an instance (or not) of a class ClassName in the root namespace
- 'namespaceName\ClassName->methodName()' represents any call of a method called methodName from an instance (or not) of a class ClassName located in the namespace namespaceName

2.5.5.3 Properties

Start your selector with a Classname (Interface or Traits will also work, inheritance is taken into account), do not end with parenthesis.

Of course, the property part of selectors are case sensitive (as properties are case sensitive in PHP).

- 'ClassName->propertyName' represent any use of a property called propertyName from an instance (or not) of a class ClassName in the root namespace
- 'namespaceName\ClassName->propertyName' represent any use of a property called propertyName from an instance (or not) of a class ClassName located in the namespace namespaceName

By default, read and write operations are considered from a property selector. To specificly hook write or read operations, prefix your selector by "read" or "write".

- 'read ClassName->propertyName' represent all the reading operations on the property propertyName of objects of type ClassName in the root namespace
- 'write ClassName->propertyName' represent all the writing operations on the property propertyName of objects of type ClassName in the root namespace

You can use both :: and -> as a seperator for classes/method class/properties (e.g. Class->method() equals Class::method()).

2.5.6 Selectors using wildcards examples

- 'startingFunctionName*()' represent any call of a function who's name starts with startingFunctionName in the root namespace
- '*endingFunctionName()' represent any call of a function who's name ends with endingFunctionName in the root namespace
- '*\functionName()' represent any call of a function called functionName in any single level namespace
- '**\functionName()' represent any call of a function called functionName in any two level namespace
- 'StartingClassName*->methodName()' represent any call of a method called methodName from an instance (or not) of a class who's name start with StartingClassName in the root namespace

- '*EndingClassName->methodName()' represent any call of a method called methodName from an instance (or not) of a class who's name end with EndingClassName in the root namespace

2.5.7 Selectors using super wildcards examples

- ***::admin*() represents every call of a method starting by admin of any class in any namespace
- ***() represents every call of any method in any namespace

Chapter 3

Advanced topics

3.1 In which orer will my pointcuts / advice will be resolved ?

The advices are executed in the registration order.

3.2 Is it possible to enable / disable the execution of aspects during runtime ?

Yes it is.

You can use the ini directive aop.enable to do so.

```
<?php
ini_set("aop.enable", "1");
echo "aop is enabled\n";
function foo () {
   echo "I'm foo\n";
}
$adviceShowFoo = function () {
   echo "After foo\n";
};

aop_add_after('foo()', $adviceShowFoo);
foo();

ini_set('aop.enable', '0');
echo "aop is now disabled\n";
foo();
echo "But you can still register new aspects\n";
```

```
aop_add_after('f*()', $adviceShowFoo);
foo();

ini_set('aop.enable', '1');
echo "Aop is now enabled\n";
foo();
```

will output

```
aop is enabled
I'm foo
After foo
aop is now disabled
I'm foo
But you can still register new aspects
I'm foo
Aop is now enabled
I'm foo
After foo
After foo
```