Go! AOP documentation Documentation

Release 0.0.1-alpha

Alexander Lisachenko

Jan 08, 2018

Table of Contents

Go! AOP is framework written in PHP which enables support for aspect oriented programming (AOP) in your PHP project. As you may already know, PHP does not supports AOP natively, and starting from PHP 5.5, AOP is not even available trough PHP extension.

In that matter, purpose of this library is to provide PHP developers with powerful development tool in form of aspect oriented programing trough userland library that can be used in almost every environment, regardless of the fact whether some framework is used in project or not.

Go! AOP is tightly coupled with Composer, so that is important library requirement: autoloading of PHP classes must be done trough Composer.

This library was inspired by Lithium PHP framework and later on, by Spring framework for Java.

CHAPTER 1

Introduction to aspect oriented programming (AOP)

Purpose of this chapter is to give you a brief introduction to aspect oriented programing as well as some theoretical background and practical examples of its usage in solving real life problems. It is advisable to read this chapter, regardless if you are familiar with aspect oriented programming paradigm or not. If you are familiar with AOP and have some experience with it, it will remind you of important features and concepts in AOP. On the other hand, of you are not (and most PHP developers aren't since AOP is not natively supported) it can provide you with neccessary basis for successful implementation of AOP concepts into your project.

1.1 Definition

From Wikipedia:

> Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code (an advice) without modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification, such as "log all function calls when the function's name begins with 'set'". This allows behaviors that are not central to the business logic (such as logging) to be added to a program without cluttering the code, core to the functionality.

Definition introduces important term, a *cross-cutting concern*, which ought to be understood in order to proceed. In that term, we have to define cross-cutting concern as well:

> Cross-cutting concerns are aspects of a program that affect other concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and can result in either scattering (code duplication), tangling (significant dependencies between systems), or both.

Aspect-oriented programming aims to encapsulate cross-cutting concerns into aspects to retain modularity.

However, these academic definitions may confuse you, especially if you do not have previous experience with AOP. Fortunately, they can be quite simplified and easily explained trough concepts with which you are already familiar with.

As previously stated in definition of aspect-oriented programming, final goal which ought to be achieved is increased modularity. Modularity is the degree to which a system's components may be separated, recombined and reused. The very origin of concept of modularity is old as civilisation itself. It is intellectual strategy as well on how to successfully handle large problem: *divide et impera* (divide and conquer). In programing, we use this strategy all the time.

So, this problem solving strategy is familiar to every developer. However, various programing languages and their implemented features provide us with various possibilities in modularization of our code. In procedural languages, fundamental unit of concern, responsibility is one procedure. In functional, it is a function. In object oriented development, it is a class. In aspect oriented development, unit of concern is one aspect.

Truth to be told, aspect oriented programming can not exists without functions, procedures, classes, methods, etc. It is a complement to known programming paradigms. However, it requires different mindset, different approach when thinking about software architecture and development problems.

If you are still confused about aspect oriented programming, no worries, read *next chapter* which will gradually, trough examples, introduce you to problems and motivations for design of aspect oriented programming and how aspect oriented programming handles those situations much better than known, usually used, tools and techniques.

1.2 Motivation for aspect oriented development

In order to understand aspect oriented development, we will consider some usual, frequent problems that we stumble upon on every day basis. We will try to solve them by employing standard, commonly used methods, well known to all developers. We will emphasize issues with appliance of those methods, their flaws and drawbacks. On top of that, we will show how aspect oriented programming can address those issues much better, removing all flows and drawbacks of other approaches from the table.

1.2.1 Design patterns

According to some authors, aspect oriented programming is implementation of Decorator, Proxy and Adapter design pattern. While that is oversimplification of aspect oriented programming, it is a good start to show its power and flexibility, since aspect oriented programming can be used to implement stated patterns much easier with more elegance. Hence, Hannemann and Kiczales demonstrated that 17 out of the 23 design patterns described in book Design Patterns - Elements of Reusable Object-Oriented Software could be simplified by simply using aspect-oriented programming.

Beside that, design patterns are good starting point for demonstration of aspect oriented programing since they are familiar to vas majority of developers as well.

Decorator

Purpose of Decorator pattern (also known as Wrapper) is to enable to extend and/or to modify behaviour of some class instance without actually modifying that class code. Decorator pattern enables us to adhere to the Single responsibility principle throughout our project. Let's consider simple banking problem that we can have in our project:

Listing 1.1: AccountTransferService.php

```php
<?php

/**
 * Service which handles account transactions
 */
class AccountTransferService
{
    /**
     * Transfer amount from one account to another.
     */
    public function transfer(Account $from, Account $to, float $amount)
    {
        $from->setBalance($from->getBalance() - $amount);
        $to->setBalance($to->getBalance() + $amount)
    }
}
```

Class above is an example of appliance of single responsibility principle, it has one method and it does single, fundamental operation. However such code is available only in examples, real life problems are much more complex.

In that matter, since this is a banking system and we are dealing with money, we need to log everything that happens with accounts. We need to modify class to support logging as well:

Listing 1.2: AccountTransferService.php

```php
<?php

/**
* Service which handles account transactions
*/
class AccountTransferService
{
    /**
     * @var Logger
     */
    private $logger;

    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * Transfer amount from one account to another.
     * Log transaction attempt and success.
     */
    public function transfer(Account $from, Account $to, float $amount)
    {
        $this->logger->log(sprintf('About to transfer amount "%s" from "%s" to "%s".',
↪$amount, $from->getAccountNumber(), $to->getAccountNumber()));

        $from->setBalance($from->getBalance() - $amount);
        $to->setBalance($to->getBalance() + $amount)

        $this->logger->log(sprintf('Successfully transferred amount "%s" from "%s" to "
↪%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber()));
    }
}
```

What happened here is that our class method got one more responsibility and in order to fulfill that task, one additional dependency - `Logger` class.

Of course, all banking transactions have to satisfy atomicity of operations, so transfer of funds must be executed within database transaction context. In that matter, we need to additionally modify our code to support such requirement:

Listing 1.3: AccountTransferService.php

```php
<?php

/**
```

```php
 * Service which handles account transactions
 */
class AccountTransferService
{
    /**
     * @var Logger
     */
    private $logger;

    /**
     * @var Database
     */
    private $database;

    public function __construct(Logger $logger, Database $database)
    {
        $this->logger = $logger;
        $this->database = $database;
    }

    /**
     * Transfer amount from one account to another.
     * Log transaction attempt and success.
     */
    public function transfer(Account $from, Account $to, float $amount)
    {
        $this->database->beginTransaction();

        $this->logger->log(sprintf('About to transfer amount "%s" from "%s" to "%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber()));

        try {
            $from->setBalance($from->getBalance() - $amount);
            $to->setBalance($to->getBalance() + $amount)
        } catch (\Exception $e) {
            $this->logger->log(sprintf('Unable to transfer amount "%s" from "%s" to "%s". Reason: "%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber(), $e->getMessage()));
            throw $e;
        }

        $this->logger->log(sprintf('Successfully transferred amount "%s" from "%s" to "%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber()));
    }
}
```

Banking systems tends to be slow (mostly because of vast amount of data which they store), so good caching is a must in order to have nice user experience.

Listing 1.4: AccountTransferService.php

```php
<?php

/**
 * Service which handles account transactions
 */
class AccountTransferService
{
    /**
     * @var Logger
     */
    private $logger;

    /**
     * @var Database
     */
    private $database;

    /**
     * @var Database
     */
    private $cache;

    public function __construct(Logger $logger, Database $database, Cache $cache)
    {
        $this->logger = $logger;
        $this->database = $database;
        $this->cache = $cache;
    }

    /**
     * Transfer amount from one account to another.
     * Log transaction attempt and success.
     */
    public function transfer(Account $from, Account $to, float $amount)
    {
        $this->database->beginTransaction();

        $this->logger->log(sprintf('About to transfer amount "%s" from "%s" to "%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber()));

        try {
            $from->setBalance($from->getBalance() - $amount);
            $to->setBalance($to->getBalance() + $amount)
        } catch (\Exception $e) {
            $this->logger->log(sprintf('Unable to transfer amount "%s" from "%s" to "%s". Reason: "%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber(), $e->getMessage()));
            throw $e;
        }

        $this->logger->log(sprintf('Successfully transferred amount "%s" from "%s" to "%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber()));

        $this->cache->save($from);
        $this->cache->save($to);
    }
}
```

Not everyone should be able to transfer money as they please, some kind of security check should be implemented as well. Banking systems are all about security.

Listing 1.5: AccountTransferService.php

```php
<?php

/**
* Service which handles account transactions
*/
class AccountTransferService
{
    /**
     * @var Logger
     */
    private $logger;

    /**
     * @var Database
     */
    private $database;

    /**
     * @var Database
     */
    private $cache;

    /**
     * @var Security
     */
    private $security;

    public function __construct(Logger $logger, Database $database, Cache $cache,␣
↪Security $security)
    {
        $this->logger = $logger;
        $this->database = $database;
        $this->cache = $cache;
        $this->security = $security;
    }

    /**
     * Transfer amount from one account to another.
     * Log transaction attempt and success.
     */
    public function transfer(Account $from, Account $to, float $amount)
    {
        if (!$this->security->isGranted('WITHDRAW', $from)) {
            throw new AccessDeniedException();
        }

        if (!$this->security->isGranted('DEPOSIT', $to)) {
            throw new AccessDeniedException();
        }

        $this->database->beginTransaction();

        $this->logger->log(sprintf('About to transfer amount "%s" from "%s" to "%s".',
↪$amount, $from->getAccountNumber(), $to->getAccountNumber()));

        try {
            $from->setBalance($from->getBalance() - $amount);
            $to->setBalance($to->getBalance() + $amount)
        } catch (\Exception $e) {
            $this->logger->log(sprintf('Unable to transfer amount "%s" from "%s" to "%s".
↪ Reason: "%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber(), $e->
↪getMessage()));
            throw $e;
        }

        $this->logger->log(sprintf('Successfully transferred amount "%s" from "%s" to "
↪%s".', $amount, $from->getAccountNumber(), $to->getAccountNumber()));

        $this->cache->save($from);
        $this->cache->save($to);
    }
}
```

So what have happened here? We started with one simple class with one simple operation implemented following single responsibility principle and we ended up with class with 4 dependencies and bloated method doing several different things, beside the operation for which method was designed for.

1.3 Glossary

1.3.1 Aspect

A modularization of a concern that cuts across multiple objects. Logging, caching, transaction management are good examples of a crosscutting concern in PHP applications. Go! defines aspects as regular classes implemented empty Aspect interface and annotated with the @Aspect annotation.

1.3.2 Join point

A point during the execution of a script, such as the execution of a method or property access.

1.3.3 Advice

Action taken by an aspect at a particular join point. There are different types of advice: @Around, @Before and @After advice.

1.3.4 Pointcut

A regular expression that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).

1.3.5 Introduction (also known as an inter-type declaration)

Go! allows you to introduce new interfaces (and a corresponding implementation with trait) to any user-defined class. For example, you could use an introduction to make all Data Transfer Objects implement an Serializable interface, to simplify persistence.

1.3.6 Weaving

Linking aspects with other application types or objects to create an advised object. This can be done at any time: compile time, load time, or at runtime. Go! performs weaving at runtime and doesn't require any additional steps to transform the source code.

CHAPTER 2

Installation

In next chapter, you will find all necessary information in regards to successful installation and configuration of Go! AOP framework into your project. Note that Go! AOP does not require any framework to be installed as well, it can be used in any PHP project, however, class autoloading and package management must be done via Composer.

Presuming that you are familiar with Composer, that Composer is globally available on your machine, first step to to start with installation of Go! AOP is to execute following console command:

```
composer require goaop/framework
```

Of course, you can require `goaop/framework` directly in your `composer.json`:

```
{
    "require": {
        "goaop/framework": "^2.1"
    }
}
```

and then, update your dependencies:

```
composer update
```

Depending on your project, read about further steps in order to configure Go! AOP to suits your needs.

2.1 Install Go! AOP framework on any PHP project

Considering that you have already installed `goaop/framework` by executing console command:

```
composer require goaop/framework
```

next thing required from you is to create application aspect kernel. That means that you have to create your own aspect kernel class, per example `ApplicationAspectKernel`, which must extends `Go\Core\AspectKernel`. This class will manage all aspects of your application in one place.

Listing 2.1: app/ApplicationAspectKernel.php

```php
<?php

use Go\Core\AspectKernel;
use Go\Core\AspectContainer;

/**
 * Application Aspect Kernel
 */
class ApplicationAspectKernel extends AspectKernel
{

    /**
     * Configure an AspectContainer with advisors, aspects and pointcuts
     *
     * @param AspectContainer $container
     *
     * @return void
     */
    protected function configureAop(AspectContainer $container)
    {
    }
}
```

2.1.1 Configuring framework

When you created your aspect kernel, you have to configure it and initialize it into your front controller.

Listing 2.2: web/index.php

```php
<?php

// First, include Composer's autoload script from your vendor directory
include __DIR__ . '/../vendor/autoload.php';

// Configure Go! AOP - see configuration references for more details and options
$aopConfig = [
    'debug'     => true,
    'cacheDir'  => __DIR__ . '/../var/cache/aop',
    'appDir' => __DIR__ . '/../src/'
];

// Create application aspect kernel
$applicationAspectKernel = ApplicationAspectKernel::getInstance();

// Initialize it with your configuration
$applicationAspectKernel->init($aopConfig);

// Rest of your front controller code goes here...
```

You will notice that it is required to load Composer first, then Go! AOP aspect kernel and then you can load your code. This is requirement of the Go! AOP framework since weaving is executed in runtime (read more about interception and weaving process here: TODO). You probably noticed that Go! AOP implements `AspectKernel` as singleton as well, which is done by design (see: *Aspect kernel is singleton implementation* for more details).

Note that not this example contains only minimum required configuration for Go! AOP to work, for all other options and details see: *Configuration references*.

2.1.2 Create and register aspect

Aspect oriented development is based on aspects and pointcuts, so in code bellow is given an example of simple aspect:

Listing 2.3: src/Aspect/MyFirstAspect.php

```php
<?php

namespace Aspect;

use Go\Aop\Aspect;
use Go\Aop\Intercept\MethodInvocation;
use Go\Lang\Annotation as Pointcut;

/**
 * My first aspect
 */
class MyFirstAspect implements Aspect
{

    /**
     * Method that will be invoked before targeted method is invoked.
     *
     * @param MethodInvocation $invocation Invocation
     * @Pointcut\Before("execution(public Example->*(*))")
     */
    protected function beforeMethodExecution(MethodInvocation $invocation)
    {
        $object    = $invocation->getThis();       // You can access object on which␣
↪method is invoked
        $arguments = $invocation->getArguments(); // You can access method␣
↪invocation arguments
        $method    = $invocation->getMethod();     // Even method metadata, and much␣
↪more...

        // And, of course, you can execute your application logic
        echo sprintf('Class "%s" method "%s" has just been invoked with %s arguments
↪', get_class($object), $method->getName(), count($arguments));
    }
}
```

In order for weaving and interception to occur and method `beforeMethodExecution()` of your aspect `Aspect\MyFirstAspect` to be executed, you have to register it in your previously created aspect kernel:

Listing 2.4: app/ApplicationAspectKernel.php

```php
<?php

use Go\Core\AspectKernel;
use Go\Core\AspectContainer;
use Aspect\MyFirstAspect;

/**
 * Application Aspect Kernel
 */
class ApplicationAspectKernel extends AspectKernel
```

```
{

    /**
     * Configure an AspectContainer with advisors, aspects and pointcuts
     *
     * @param AspectContainer $container
     *
     * @return void
     */
    protected function configureAop(AspectContainer $container)
    {
        $container->registerAspect(new MyFirstAspect());
    }
}
```

2.2 Install Go! AOP on Symfony framework

Installation and configuration of Go! AOP on Symfony project is done trough Symfony bundle `goaop/goaop-symfony-bundle`, which you can install with composer:

```
composer require goaop/goaop-symfony-bundle
```

You do not have to worry about framework installation, bundle will pull down all required dependencies.

Next, you have to load bundle into your Symfony application kernel:

Listing 2.5: app/ApplicationAspectKernel.php

```
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = [
            // Go! AOP bundle as first loaded bundle
            new Go\Symfony\GoAopBundle\GoAopBundle(),


            // ... any other bundle that is being used by project
        ];

        return $bundles;
    }
}
```

IMPORTANT NOTE: Go! AOP bundle must be first item in bundle list. Failing to do so will cause an exception, since Go! AOP framework requires it to be loaded first in order for engine to work correctly.

2.2.1 Configuring framework

Configuring Go! AOP framework uses Symfony configuration system, so you can configure it as any other Symfony bundle. In code below is default bundle configuration with initial values sufficient enough to accommodate majority of Symfony projects.

Listing 2.6: app/config/config.yml

```yaml
go_aop:
    cache_warmer: true
    doctrine_support: false
    options:
        debug: '%kernel.debug%'
        app_dir: '%kernel.root_dir%/../src'
        cache_dir: '%kernel.cache_dir%/aspect'
        include_paths: []
        exclude_paths: []
        exclude_paths: []
        features: []
```

Note that in this example configuration parameters under `options` key are options related to Go! AOP framework, for details see: *Configuration references*. While Go AOP! framework uses camel case (`camelCase`) notation for configuration parameters, Symfony bundle uses Symfony's configuration notation, underscore, for same configuration keys.

However, `cache_warmer` and `doctrine_support` are configuration options available only for Symfony bundle, and therefore, their details are provided bellow:

- `cache_warmer`, boolean, default `true`. Enables or disables an automatic AOP cache warming within the application. By default, cache warming is enabled (recommended). You may disable it only if you have serious issues with cache warming process.
- `doctrine_support`, boolean, default `false`. Experimental, alpha. Due to mapping issues caused by weaving of entities (as explained here: *Weaving of Doctrine entities*) it is not possible to weave Doctrine entities. This option will turn on basic support for weaving Doctrine entities, however, support is experimental, incomplete, and not ready for production.

XML configuration support

XML format is supported for configuration of this bundle as well. In order to validate your XML configuration, you may use XML Schema file available on following url: https://github.com/goaop/goaop-symfony-bundle/blob/master/Resources/config/schema/configuration-1.0.0.xsd. This is more convenient configuration method since it is possible, trough configuration of IDE, to get intelli sense and autocomplete support.

Example of XML configuration is given bellow:

Listing 2.7: app/config/go-aop.xml

```xml
<?xml version="1.0" ?>
<container
        xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:go-aop="http://go.aopphp.com/xsd-schema/go-aop-bundle"
        xsi:schemaLocation="http://symfony.com/schema/dic/services
                            http://symfony.com/schema/dic/services/services-
↪1.0.xsd
                            http://go.aopphp.com/xsd-schema/go-aop-bundle
                            http://go.aopphp.com/xsd-schema/go-aop-bundle/
↪configuration-1.0.0.xsd">
```

```
    <go-aop:config>

        <go-aop:cache-warmer>true</go-aop:cache-warmer>

        <go-aop:doctrine-support>false</go-aop:doctrine-support>

        <go-aop:options>

            <go-aop:debug>true</go-aop:debug>
            <go-aop:app-dir>%kernel.root_dir%/../src</go-aop:app-dir>
            <go-aop:cache-dir>%kernel.cache_dir%/aspect</go-aop:cache-dir>

            <go-aop:include-path>/path/to/include/directory</go-aop:include-
↪path>
            <go-aop:include-path>/other/path/to/include/directory</go-
↪aop:include-path>

            <go-aop:exclude-path>/path/to/directory/to/exclude</go-
↪aop:exclude-path>
            <go-aop:exclude-path>/other/directory/path/to/exclude</go-
↪aop:exclude-path>

            <go-aop:container-class>Container\Class</go-aop:container-class>

            <go-aop:feature>INTERCEPT_FUNCTIONS</go-aop:feature>
            <go-aop:feature>INTERCEPT_INCLUDES</go-aop:feature>
            <go-aop:feature>INTERCEPT_INITIALIZATIONS</go-aop:feature>

        </go-aop:options>

    </go-aop:config>

</container>
```

2.2.2 Create and register aspect

Aspects are services in the Symfony application and loaded into the aspect container with the help of compiler pass that collects all services tagged with `goaop.aspect` tag.

Considering that you have, per example, following aspect:

Listing 2.8: app/ApplicationAspectKernel.php

```php
<?php

namespace App\Aspect;

use Go\Aop\Aspect;
use Go\Aop\Intercept\MethodInvocation;
use Go\Lang\Annotation as Pointcut;
use Psr\Log\LoggerInterface;

/**
 * Logs every successful method execution
 */
class LoggingAspect implements Aspect
{
    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    /**
     * Method that will be invoked after targeted method is invoked.
     *
     * @param MethodInvocation $invocation Invocation
     * @Pointcut\After("execution(public **->*(*))")
     */
    protected function afterMethodExecution(MethodInvocation $invocation)
    {
        $object    = $invocation->getThis();      // You can access object␣
↪on which method is invoked
        $arguments = $invocation->getArguments(); // You can access method␣
↪invocation arguments
        $method    = $invocation->getMethod();    // Even method metadata,␣
↪and much more...

        $this->logger->info('Successfully executed method {name} of class
↪{class} with {count} arguments.', [
            '{name}'    => $method->getName(),
            '{class}'   => get_class($object),
            '{count}'   => count($arguments),
            'arguments' => $arguments,
        ]);
    }
}
```

only thing required for aspect to be registered is to be registered as service in Symfony framework and tagged with `goaop.aspect` tag:

Listing 2.9: app/config/config.yml

```yaml
services:
    logging.aspect:
        class: App\Aspect\LoggingAspect
        arguments: ['@logger']
        public: false
        tags:
            - { name: goaop.aspect }
```

Note that aspects may be registered as `private` services, which can contribute to optimization of your service container.

2.3 Install Go! AOP on Laravel framework

2.4 Install Go! AOP on Zend framework

2.5 Install Go! AOP Yii framework

Considering that you have already installed `goaop/framework` by executing console command:

```
composer require goaop/framework
```

for versions prior to 2.1 it is required to modify `composer.json` file:

```
{
   "name": "yii/project",
      "require": {
         "yiisoft/yii2": "^2.0",
         "goaop/framework": "^2.1"
      },
      "autoload": {
         "psr-4": {
             "api\\": "api",
             "common\\": "common",
             "console\\": "console"
         }
      }
   }
}
```

Don't forget to dump autoload every time when you modify `composer.json` file:

```
composer dumpautoload
```

Create your aspect kernel class (make sure it can be autoloaded via Composer) and your aspects:

Listing 2.10: YiiAspectKernel.php

```php
<?php

use Go\Core\AspectContainer;
use Go\Core\AspectKernel;

 class YiiAspectKernel extends AspectKernel
 {

    protected function configureAop(AspectContainer $container)
    {
       $container->registerAspect(new MyAspect());
    }
 }
```

And finally, you will have to modify your `{app}/web/index.php` file:

Listing 2.11: YiiAspectKernel.php

```php
 <?php

 use YiiAspectKernel;
 use common\config\ConfigLoader;
 use yii\web\Application;

 require __DIR__ . '/../../vendor/autoload.php';

 defined('YII_DEBUG') or define('YII_DEBUG', in_array(getenv('YII_DEBUG'), ['true', '1
↪']));
 defined('YII_ENV') or define('YII_ENV', getenv('YII_ENV'));

```

```
// Initialize an application aspect container
ApiAspectKernel::getInstance()->init([
    'debug'     => YII_DEBUG,
    'appDir'    => __DIR__ . '/../../',
    'cacheDir' => __DIR__ . '/../runtime/aspect',
    'excludePaths' => [
        __DIR__ . '/../runtime/aspect',
        __DIR__ . '/../../vendor',
    ],
]);

require __DIR__ . '/../../vendor/yiisoft/yii2/Yii.php';
spl_autoload_unregister(['Yii', 'autoload']);
require __DIR__ . '/../../common/config/bootstrap.php';
require __DIR__ . '/../config/bootstrap.php';

$config = ConfigLoader::load('api');

$application = new Application($config);
$application->run();
```

Most important thing in code here which you should notice is `spl_autoload_unregister(['Yii', 'autoload'])` which will disable Yii's autoloader and use Composer's one instead, which enables to Go! AOP to intercept code class loading and execute weaving.

2.6 Configuration references

- `debug`: Optional, boolean. Denotes whether aspect kernel operates in debug mode. By default, this parameter is set to `false`. It is advisable to set this parameter to `true` when developing application.
- `appDir`: Optional, string. Absolute path to your application root directory. If not provided, kernel will try to guess your application directory based on location of `vendor` directory where framework is installed. It is advised to set this location manually. Files in this directory will be scanned and analyzed for weaving.
- `cacheDir`: Required, string. Path to cache location where aspect kernel will store its output in order to optimize load time.
- `cacheFileMode`: Optional, integer. Binary mask of permission bits that is set to cache files. By default, value is set to `0770 & ~umask()`.
- `annotationCache`: Optional, object. Implementation of `Doctrine\Common\Cache\Cache` interface which will be used as cache driver for Doctrine's annotation reader. If not provided, `Doctrine\Common\Cache\FilesystemCache` is used with cache location defined within your `cacheDir` configuration, inside `_annotations` directory.
- `includePaths`: Optional, array of strings. Beside application directory (`appDir` configuration parameter), you may state additional directories for scanning and analysis by aspect kernel for weaving.
- `excludePaths`: Optional, array of strings. You may provide a list of directories which you would like to be omitted from weaving.
- `containerClass`: Optional, string. Full qualified class name of aspect container implementation, that is, implementation of `Go\Core\Container`. If not provided, `Go\Core\GoAspectContainer` is used by default. Its intention is to allow you to provide your own implementation of aspect container, or modified version of default one.

- `features`: Optional, integer, default `0`. Binary mask of included features. Feature supported by Go! AOP are defined within `Go\Aop\Features` interface:
  - `Go\Aop\Features::INTERCEPT_FUNCTIONS` - Enables interception of functions. This feature can have noticeable impact on performances of weaving.
  - `Go\Aop\Features::INTERCEPT_INITIALIZATIONS` - This feature enables interception of "new" operator in the source code. As function interception, this feature might have noticeable impact on performances of weaving.
  - `Go\Aop\Features::INTERCEPT_INCLUDES` - Enables interception of `include` and `require` operators in legacy code. It's usage is depreciated and support might be dropped in the future. Use Composer for autoloading classes instead.
  - `Go\Aop\Features::PREBUILT_CACHE` - Useful within read-only filesystem, denotes that classes are already weaved and class cache is already built.
  - `Go\Aop\Features::PARAMETER_WIDENING` - This feature will instruct weaving engine to skip type declarations when generating weaved methods. That would allow you, per example, to easily implement adapter pattern. However, do note that **this feature is only available for PHP of version 7.2.0 or higher**. For details, see: **'https://wiki.php.net/rfc/parameter-no-type-variance<https://wiki.php.net/rfc/parameter-no-type-variance>'_**.

2.6.1 Annotation cache

Go! AOP uses Doctrine's annotation reader for reading and analyzing class annotations. By default, annotation cache is stored on local filesystem, since `Doctrine\Common\Cache\FilesystemCache` is used. This is a reasonable setting for most of the use case scenarios.

However, there are use cases where default implementation could not be used, or it is not appropriate to use it:

- **Testing environment**: For your unit tests or functional tests, it is advisable to use instance of `Doctrine\Common\Cache\ArrayCache` since it does not leave any footprint on your filesystem and memory.
- **Read-only filesystems**: If you are deploying your application to read-only filesystem (like PHAR archive or similar) filesystem based cache is not option. In that case you may use any other implementation, such as APCu, Redis, Memcache or similar. If nothing of available does not works for you, you can always sacrifice some performances and fallback to `Doctrine\Common\Cache\ArrayCache`

CHAPTER 3

Using Go! AOP

3.1 Understanding aspects

In order to understand aspects, we have to go back to fundamental concept of modularity in programming, that is, appliance of *divide at impera* strategy, which helps us to split complex problems to smaller ones and solve them as such, one by one.

Every programing paradigm defines modularity differently.

For procedural programing languages, fundamental unit of modularity is one single procedure, that is, complex problem is broken down to smaller, simpler ones, and each procedure should solve such smaller sub-problem. Same goes to functional programing languages, unit of modularity for functional languages is function.

Following same logic, it can be deducted that unit of modularity for object oriented languages is a class. Although such statement can be subject of argument (since class can consist of several methods, solving various problems in same time), in ideal world following Single responsibility principle a class methods will be organized around single sub-problem.

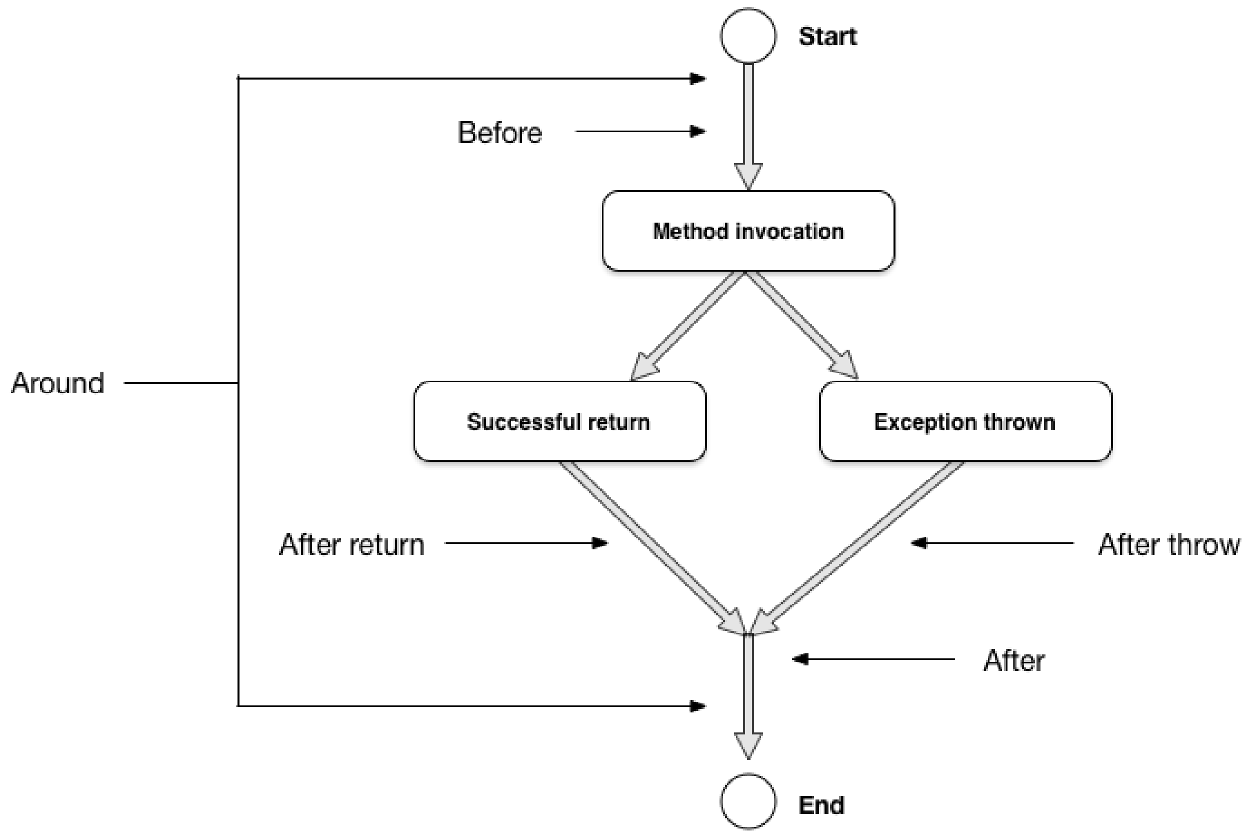
In aspect oriented programing paradigm, aspect is unit of modularity. As in above mentioned, aspect suppose to solve one simple sub-problem. The difference is in the way how aspect approaches to the subject. Namely, aspect oriented programing defines a cross-cutting concerns.

> In aspect-oriented software development, cross-cutting concerns are aspects of a program that affect other concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and can result in either scattering (code duplication), tangling (significant dependencies between systems), or both.
>
> —From Wikipedia

3.2 Types of advices

In aspect oriented programming, pointcut answers on question which operation (method, function, initialization, etc.), that is, answers on question "**what do you want to intercept?**". On the other hand, advice defines "**how do you want to intercept"**, that is, specifies the particular point of method invocation which should be intercepted.

Consider image below. It shows method invocation lifecycle, from its start to end. When method is invoked, the result of invocation can be either successful (it returns some value, or exists returning nothing, a *void*), or exception is thrown.



Having in mind a flow and lifecycle of method invocation, aspect oriented programming defines 5 fundamental types of advices, that is, 5 points of method invocation interception points as shown in image above. You can intercept in any of those points and execute your arbitrary code.

Depending of used advice, possibility varies, which are explained in following text.

3.2.1 Before advice

Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

3.2.2 After returning advice

Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

3.2.3 After throwing advice

Advice to be executed if a method exits by throwing an exception.

3.2.4 After (finally) advice

Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

3.2.5 Around advice

Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

3.3 Pointcut syntax

In aspect oriented programming, a pointcut is a regular expression that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).

In simpler terms, pointcut answers on question **"what do you want to intercept?"**

Different frameworks support different pointcut expressions, while Java's AspectJ is considered as industry standard. Go! AOP does not diverts from standards established by AspectJ, so any developer familiar with Java and AspectJ will find itself within familiar grounds when working with Go! AOP.

However, there are certain differences, possibilities as well as limitations within PHP language itself. Go! AOP had to take into consideration those differences when adopting AspectJ standard.

3.3.1 Methods and Constructors

You may intercept every `public` and/or `protected` method of class, as well as constructor (which is also a class method with specific properties - it is invoked at class initialization and it does not returns value). Subject of interception can be final and/or static class method as well.

Pointcut expression pattern is: `execution([VISIBILITY_MODIFIER] [CLASS_NAME_FILTER][TYPE_OF_INVOCATION][METHOD_NAME_FILTER](*))`

- `execution` keyword denotes that class method (class method or static class method) is subject of interception.
- With `[VISIBILITY_MODIFIER]` you may denote method visibility modifier. Go! AOP supports interceptions of `public` and `protected` methods. If you want to intercept both `public` as well as `protected` methods, you may use "pipe" (`|`) operator to denote both, in example: `public|protected`.
- `[CLASS_NAME_FILTER]` allows you to specify expression that will be used to match full qualified class name which ought to be intercepted. Expression wildcards which can be used for namespace matching are simplified and limited, however, sufficient enough for everyday usage:
  - `*` (astrix, star) matches any character and digit in namespace part. Its regular expression equivalent is `[^\\\\]+` which matches any character as many time as possible between namespace separators (`\` - backslash).
  - `**` (double astrix) matches any namespace. Its regular expression equivalent is `.+` which matches any character as many times as possible.
  - `?` (question mark) matches any single character. Its regular expression equivalent is `.` (dot) sign.
  - `|` (pipe), which is logical `or` operator in this context. Its regular expression equivalent is pipe as well.

- `[TYPE_OF_INVOCATION]` allows you to specify whether you are matching class method or static class method. It may be `::` (*Paamayim Nekudotayim*) or `->` (method invocation). It is not possible to match booth class method and static class method within the same expression.
- `[METHOD_NAME_FILTER]` allows you to specify method name. Wildcards that can be used in method name matching are `*` (astrix, star), `?` (question mark) and `|` (pipe) operator. They share same semantics with class name filter.
- `(*)` is constant, static part of the expression, which matches any number, name and type of method arguments. AspectJ, per example, supports matching based on method arguments as well. In Go! AOP, that is not possible.

Methods via annotations

Annotation matching are another way to intercept any protected and/or public class method, as well as static class method. Pointcut expression pattern is: `@execution([ANNOTATION_FULL_QUALIFIED_CLASS_NAME])`. Note that wildcards are not supported. Only annotated public and protected methods can be matched.

- `@execution` keyword denotes that annotated class method (class method or static class method) is subject of interception.
- `[ANNOTATION_FULL_QUALIFIED_CLASS_NAME]` allows you to specify full qualified class name of your annotation which annotates method that ought to be intercepted

Examples

- `execution(public Example->method(*))` - Every execution of public method with the name `method` in the class `Example`
- `execution(public Example->method1|method2(*))` - Every execution of one of the public methods: `method1` or `method2` in the class `Example`
- `execution(public|protected Example\Aspect\*->method*(*))` - Execution of public or protected methods that have `method` prefix in their names and that methods are also within `Example\Aspect` sub-namespace. Note that only one, single namespace part will be matched.
- `execution(public **::staticMethod(*))` - Every execution of any public static method `staticMethod` in every namespace (except global one).
- `@execution(Demo\Annotation\Cacheable)` - Every execution of any method that has `Demo\Annotation\Cacheable` annotation in its docBlock.

3.3.2 Property access

Access to public and protected class properties may be intercepted as well. Note that interception of static class properties is not supported.

Pointcut expression pattern is: `access([VISIBILITY_MODIFIER] [CLASS_NAME_FILTER]->[PROPERTY_NAME_FILTE`

- `access` keyword denotes that class property is subject of interception.
- `[VISIBILITY_MODIFIER]` denotes property visibility modifier. Go! AOP supports interceptions of `public` and `protected` properties. Interception of both `public` and `protected` properties is possible with "pipe" (`|`) operator.
- `[CLASS_NAME_FILTER]` allows you to specify expression that will be used to match full qualified class name which ought to be intercepted. It has same syntax as method interception, which means that you can use same wildcards (`*`, `**`, `?`, `|`) in your expressions.

- `[PROPERTY_NAME_FILTER]` allows you to specify property name. You may use `*`, `?` and `|` wildcards for property matching.

Property access via annotations

Annotation matching is supported for class properties as well. Pointcut expression pattern is: `@access([ANNOTATION_FULL_QUALIFIED_CLASS_NAME])`. Wildcards are not supported. Only annotated public and protected properties can be matched.

- `@access` keyword denotes that annotated class property is subject of interception.
- `[ANNOTATION_FULL_QUALIFIED_CLASS_NAME]` allows you to specify full qualified class name of your annotation which annotates property that ought to be intercepted

Examples

- `access(public Example\Demo->test)` - Every access (read and write) to the public property `test` in the class `Example\Demo`.
- `access(public|protected Example\Aspect\*->fieldName)` - Every access (read and write) to a public or protected property `fieldName` which belongs to a classes inside `Example\Aspect` sub-namespace.
- `access(protected **->someProtected*Property)` - Every access to the protected properties with names that match `someProtected*Property` pattern in every class.
- `@access(Demo\Annotation\Cacheable)` - Every access to the property (read and write) that has `Demo\Annotation\Cacheable` annotation in the docBlock.

3.3.3 Initialization

With Go! AOP, you may intercept object initialization. Pointcut expression pattern is `initialization([FULL_QUALIFIED_CLASS_NAME_FILTER])`. All before mentioned wildcards are supported.

- `initialization` keyword denotes that object initialization will be intercepted.
- `[FULL_QUALIFIED_CLASS_NAME_FILTER]` allows you to specify expression that will be used to match name of class which initialization ought to be intercepted.

Have in mind that interceptions of class initialization can have noticeable impact on weaving performance since it requires scanning and analysing source code of every class which is within configured source/include directory. Therefore, this feature is disabled by default. You have to configure your aspect kernel to include this feature. For details see: *Configuration references*.

Static initialization

Static initialization allows you to execute arbitrary code when some class or classes are about to be loaded for the first time in application execution life cycle, that is, when autoloader loads those classes into memory. Pointcut expression pattern is `staticinitialization([FULL_QUALIFIED_CLASS_NAME_FILTER])`. All before mentioned wildcards are supported.

- `staticinitialization` keyword denotes that first time class loading into memory is intercepted.
- `[FULL_QUALIFIED_CLASS_NAME_FILTER]` allows you to specify expression that will be used to match name of class which static initialization ought to be intercepted.

Examples

- `initialization(Demo\Example)` - Every initialization of class instance of `Demo\Example`, that is, when `new Demo\Example()` is invoked.
- `initialization(Demo\**)` - Every initialization of any class instance within `Demo` sub-namespaces.
- `staticinitialization(Demo\Example)` - First time when the class `Demo\Example` is loaded into the memory of application execution context.
- `staticinitialization(Demo\**)` - First time of loading of any class within `Demo` sub-namespace into the memory application execution context.

3.3.4 Function execution

Go! AOP supports interception of system functions. Syntax is similar to class method interception, which means that you can use same wildcards (`*`, `**`, `?`, `|`) for matching as well. Note that this feature is not common among other programming languages with support for aspect oriented programming.

Pointcut expression pattern is `execution([NAMESPACE_FILTER]\[FUNCTION_NAME_FILTER](*))`

- `execution` keyword denotes that function execution will be intercepted.
- `[NAMESPACE_FILTER]` allows you to specify expression that will be used to match namespace in which system function will be intercepted.
- `[FUNCTION_NAME_FILTER]` allows you to specify expression that will be used to match system function name which will be intercepted.
- `(*)` is constant, static part of the expression, which matches any number, name and type of method arguments.

Have in mind that interceptions of functions can have noticeable impact on weaving performance since it requires scanning and analysing source code of every class which is within configured source/include directory. Therefore, this feature is disabled by default. You have to configure your aspect kernel to include this feature. For details see: *Configuration references*.

Examples

- `execution(**\file_get_contents())` - Every execution of system function “file_get_contents” within all namespaces
- `execution(Example\Aspect\array_*(*))` - Every execution of any system function “array_*” within “ExampleAspect” namespace

3.3.5 Complex expressions

With pointcut expressions you can specify which methods, properties, functions you would like to intercept. Beside wildcards within pointcut expressions that are already mentioned, Go! AOP supports some basic lexical expressions, as well as portion of Boolean algebra. That allows you to build quite complex and powerful expressions to mach desired subjects of intercepting.

Supported wildcards

For sake of completeness, we are going to state all supported wildcards here which can be used (where applicable) to match part of the name within pointcut expressions

- `*` (astrix, star) matches any character and digit in name part. Its regular expression equivalent is `[^\\\\]+` which matches any character as many time as possible between namespace separators (`\` - backslash).
- `**` (double astrix) matches any namespace. Its regular expression equivalent is `.+` which matches any character as many times as possible.
- `?` (question mark) matches any single character. Its regular expression equivalent is `.` (dot) sign.
- `|` (pipe), which is logical `or` operator in this context. Its regular expression equivalent is pipe as well.

Supported operators

Basic boolean operators are supported as well:

- `!` (exclamation mark) logical negation
- `&&` (double ampersand) logical conjunction
- `||` (double pipe) logical disjunction

Lexical pointcuts

Go! AOP supports only two lexical statements, `within` and `@within`.

- `within([CLASS_NAME_FILTER])` - every property access, method execution, initialization within `[CLASS_NAME_FILTER]` class.
- `@within([ANNOTATION_CLASS_NAME_FILTER])` - every property access, method execution, initialization within class that has `[ANNOTATION_CLASS_NAME_FILTER]` annotation in the docBlock.

With `within` and/or `@within` statements, you may narrow down set of matching subjects to certain classes and/or classes with stated annotation.

Examples

- `!execution(public **->*(*))` - every execution of any method that is not public.
- `execution(* Demo->*(*)) && execution(public **->*(*))` - every execution of public method in the class `Demo`.
- `access(public Demo->foo) || access(public Another->bar)` - every access to the properties `foo` of class `Demo` or property `bar` of class `Another`.
- `(access(* Demo->*) || access(* Another->*)) && access(public **->*)` - every access to public property of class `Demo` or class `Another`. Note parentheses which can be used to establish order of operator precedence.
- `execution(public **->*(*)) && @within(Demo\Cacheable)` - every public method of class annotated with `Demo\Cacheable` annotation.

CHAPTER 4

Pointcuts and advices

Listing 4.1: somecode.php

```php
<?php

/**
 * Method that should be called before real method
 *
 * @param MethodInvocation $invocation Invocation
 * @Before(pointcut="examplePublicMethods()")
 */
 public function beforeMethodExecution(MethodInvocation $invocation)
 {
     $obj = $invocation->getThis();
     echo 'Calling Before Interceptor for method: ',
     is_object($obj) ? get_class($obj) : $obj,
     $invocation->getMethod()->isStatic() ? '::' : '->',
     $invocation->getMethod()->getName(),
     '()',
     ' with arguments: ',
     json_encode($invocation->getArguments()),
     "<br>\n";
 }
```

CHAPTER 5

Internals explained

CHAPTER 6

Limitations and known issues

6.1 Limitations

6.1.1 Aspect kernel is singleton implementation

Aspect kernel is singleton implementation by design because there is no reason to run one single application with two loaded aspect kernel with two different configurations. One class/function can be weaved only once and loaded into execution memory.

However, if application has more than one front controller, and each controller has different requirements conflicting with another, it is possible to setup different aspect kernels with different configurations for each of those front controllers.

6.1.2 Serialization/deserialization and reflections are sensitive to weaving

As have been mentioned several times, Go! AOP executes code weaving in runtime by introducing proxy class. Proxy class adds additional level of inheritance of your class, and therefore, previous assumption on which serialization/deserialization and reflective code is built upon becomes wrong.

This issue is elaborated in details here: *Serialization/deserialization and reflections are sensitive to weaving* where you can find guidelines how to prevent possible issues in your code.

Serialization/deserialization and reflections are sensitive to weaving

In order to explain issues in regards to serialization/deserialization and reflections that can be caused by weaving with Go! AOP framework, we will consider a simple example where we have a *Person* class:

Listing 6.1: src/Application/Model/Person.php

```php
<?php

namespace Application\Model;
```

```

class Person
{
    const MALE = 'male';
    const FEMALE = 'female';
    const UNDEFINED = 'undefined';

    private $id;

    private $gender;

    private $firstName;

    private $lastName;

    public function __construct($id, $gender, $firstName, $lastName)
    {
        $this->id        = $id;
        $this->gender    = $gender;
        $this->firstName = $firstName;
        $this->lastName  = $lastName;
    }

    public function getFullName()
    {
        return $this->firstName.' '.$this->lastName;
    }
}
```

Now, let's say that we have in our code introspection of class above with reflections, as well as we are serializing and deserializing instances of it:

Listing 6.2: src/application.php

```php
<?php

use \Application\Model\Person;

$person             = new Person(1, Person::UNDEFINED, 'John', 'Doe');
$reflectionProperty = new \ReflectionProperty(Person::class, 'firstName');

$reflectionProperty->setAccessible(true);
$reflectionProperty->setValue($person, 'Bob');

file_put_contents(__DIR__.'/../var/data/person.dat', serialize($person));

// ... more code here....

$person = deserialize(file_get_contents(__DIR__.'/../var/data/person.dat'));
```

If we introduce new aspect that will intercept execution of, per example, *Person::getFullName()* method:

Listing 6.3: src/Aspect/PersonAspect.php

```php
<?php

namespace Aspect;

```

```php
use Go\Aop\Aspect;
use Go\Aop\Intercept\MethodInvocation;
use Go\Lang\Annotation as Pointcut;

/**
 * Person aspect
 */
class PersonAspect implements Aspect
{

    /**
     * Add title to full name.
     *
     * @param MethodInvocation $invocation Invocation
     * @Pointcut\After("execution(public Application\Model\Person->
↪getFullName(*))")
     */
    protected function afterGetFullName(MethodInvocation $invocation)
    {
        $object    = $invocation->getThis();
        $method    = $invocation->getMethod();
        $arguments = $invocation->getArguments();

        // And, of course, you can execute your application logic
        echo sprintf('Class "%s" method "%s" has just been invoked with %s␣
↪arguments', get_class($object), $method->getName(), count($arguments));
    }
}
```

instead of original class, after weaving process, we will get two classes instead that will be loaded and used in our application, a proxied class:

Listing 6.4: var/cache/aop/Application/Model/Person.php

```php
<?php

namespace Application\Model;

class Person__AopProxied
{
    const MALE = 'male';
    const FEMALE = 'female';
    const UNDEFINED = 'undefined';

    private $id;

    private $gender;

    private $firstName;

    private $lastName;

    public function __construct($id, $gender, $firstName, $lastName)
    {
        $this->id        = $id;
        $this->gender    = $gender;
        $this->firstName = $firstName;
        $this->lastName  = $lastName;
    }

    public function getFullName()
    {
        return $this->firstName.' '.$this->lastName;
    }
}

include_once AOP_CACHE_DIR . '/_proxies/Application/Model/Person.php';
```

and, of course, a proxy class:

Listing 6.5: var/cache/aop/_proxies/Application/Model/Person.php

```php
<?php

namespace Application\Model;

class Person extends Person__AopProxied implements \Go\Aop\Proxy
{

    /**
     * Property was created automatically, do not change it manually
     */
    private static $__joinPoints = [
        'method' => [
            'getFullName' => [
                'advisor.Aspect\\PersonAspect->afterGetFullName'
            ]
        ]
    ];


    public function getFullName()
    {
        return self::$__joinPoints['method:getFullName']->__invoke($this);
    }

}
\Go\Proxy\ClassProxy::injectJoinPoints(Person::class);
```

When we go to our previous code that uses reflections to introspect `Person` class, knowing what is the end result of weaving, it is quite understandable why it does not work as it used to do:

```php
<?php

$reflectionProperty = new \ReflectionProperty(Person::class, 'firstName');

$reflectionProperty->setAccessible(true);
$reflectionProperty->setValue($person, 'Bob');
```

Class `Person` **does not define** `firstName` property anymore, that is defined in `Person__AopProxied` class. Same goes to serialization/deserialization, every instance serialized *before* weaving can not be deserialized *after* waeving anymore because `Person` class before weaving is not the same class after weaving.

How to handle serialization/deserialization of weaved classes?

First of all, it is not recommended to use default PHP serialization/deserialization for durable persistence of objects/entities in your application. Developers tend to use it, per example, when they need some schema-less persistence of object but they use relational database as storage, so they serialize object and store it in one table field.

Much better solution, if such serialization is required, is to take over control from default PHP serialization/deserialization and provide your own implementation, per example:

Listing 6.6: src/Application/Model/Person.php

```php
<?php

namespace Application\Model;

class Person
{
    const MALE = 'male';
    const FEMALE = 'female';
    const UNDEFINED = 'undefined';

    private $id;

    private $gender;

    private $firstName;

    private $lastName;

    public function __construct($id, $gender, $firstName, $lastName)
    {
        $this->id        = $id;
        $this->gender    = $gender;
        $this->firstName = $firstName;
        $this->lastName  = $lastName;
    }

    public function getFullName()
    {
        return $this->firstName.' '.$this->lastName;
    }

    /**
     * Serializes object values to JSON
     */
    public function toJson()
    {
        $data = [
            'id'        => $this->id,
            'gender'    => $this->gender,
            'firstName' => $this->firstName,
            'lastName'  => $this->lastName,
        ];

        return json_encode($data);
    }

    /**
     * Restore object values from JSON
     */
    public function fromJson($json)
    {
        $data = json_decode($json);

        $this->id        = $data['id'];
        $this->gender    = $data['gender'];
        $this->firstName = $data['firstName'];
        $this->lastName  = $data['lastName'];
    }
}
```

Of course, there are much better ways and libraries which can help you to provide custom serialization/deserialization of objects. However, the general idea is that you have full control over that process. That control is required in order to have possibility to adapt/modify code to support deserialization of previously serialized object - even after weaving.

How to handle reflections of weaved classes?

As shown in previous section, general idea is in taking control in your hands, so when the weaving kicks in, you can easily modify code to support new class hierarchy.

Same goes to reflections, instead of using PHP system classes, use your own implementation as proxy to default one:

Listing 6.7: src/Application/System/ReflectionClass.php

```php
<?php

namespace Application\System;

class ReflectionClass extends \ReflectionClass
{
}
```

By using your own modified classes for reflections, you are able to modify its code and move the introspection one level up to proxied classes when introspecting weaved classes.

6.1.3 Class weaving constrained to PSR-0 and PSR-4 compatible classes only

Go! AOP supports only weaving of classes which are compatible with PSR-0 and PSR-4 autoloading, which means that there can be only one class per file.

6.1.4 Interception of system functions is sensitive to namespace prefixing

Go! AOP supports interception and weaving of system functions, however, it is only possible if functions are referenced without namespace prefix. That means that, per example, it is possible to intercept `array_merge()` function, if function is used like stated above. However, if namespace prefix is used, per example `\array_merge()`, interception of same function is not possible.

However, developers almost never prefixes system functions with default namespace.

6.2 Known issues

6.2.1 Weaving of native PHP classes

It is not possible (yet) to weave native PHP classes and classes loaded via PHP extensions. Therefore, your point cut should not target those classes. Reason for that is that those classes are not loaded via Composer into memory, and therefore their loading process can not be intercepted by Go! AOP.

6.2.2 Weaving of classes/functions from PHAR archive

Although this might be supported in future, for now, it is not possible. However, this is not in priority list, almost every library that is available as PHAR archive is available via Composer as well. If you need weaving of those classes, use Composer to acquire library source code, not PHAR archive.

Common occurrence of this limitation is when, per example, PHPUnit is being used as PHAR archive in project for unit tests, and directory where test classes residue are not excluded from weaving. If you do not need weaving of your unit tests, it is much wiser to exclude tests from weaving (see *Configuration references*) rather than include source code of above mentioned PHPUnit library.

6.2.3 Weaving of Doctrine entities

Weaving of Doctrine entities is at the moment experimental, but unsupported due to difficulty of implementation of full support. Namely, Go! AOP injects one level of inheritance level to entities trough proxy classes modifying mapping metadata of entities. With proxy, mapped class properties are moved one level up to proxied classes. Current efforts to support weaving of Doctrine entities is focused on modifying metadata by introducing proxied classes as mapped superclasses. However, it is not stable yet and can not be used in production.

CHAPTER 7

References and resources

genindex

Index