PHP Reflect Book

Laurent Laville

PHP Reflect Book

Laurent Laville

Table of Contents

List of Examples

This complete guide documents PHP Reflect 2.6.3, published on 2015-04-15.

Part I. Getting Started

Chapter 1. Download

We distribute a PHP Archive [http://www.php.net/phar] (PHAR) that contains all required dependencies of PHP Reflect bundled in a single file.

Download the latest version [http://bartlett.laurent-laville.org/get/phpreflect-2.6.3.phar]

Make it executable, and put it into your `$PATH`.

```
$ chmod +x phpreflect-2.6.3.phar
$ mv phpreflect-2.6.3.phar /usr/local/bin/phpreflect

$ phpreflect --version
```

You can also immediately use the PHAR after you have downloaded it.

```
$ wget http://bartlett.laurent-laville.org/get/phpreflect-2.6.3.phar
$ php phpreflect-2.6.3.phar --version
```

With both methods then you have this output :

```
phpReflect version 2.6.3
```

Other alternative installations are possible. Please refer to the Chapter 7, *Installation* for details on how to do this.

Chapter 2. Configuration

With the minimalist JSON file `phpreflect.json`.

```
{
    "source-providers": [
        {
            "in": ". as current",
            "name": "/\\.(php|inc|phtml)$/"
        }
    ],
    "plugins": [
        {
            "name": "Analyser",
            "class": "Bartlett\\Reflect\\Plugin\\Analyser\\AnalyserPlugin"
        }
    ],
    "analysers" : [
        {
            "name": "Structure",
            "class": "Bartlett\\Reflect\\Analyser\\StructureAnalyser"
        }
    ]
}
```

Put it in your project's folder. Alternative locations are possible. Please refer to the Chapter 8, *The Json Configuration File* for details on how to do this.

Chapter 3. Structure

source-providers
: this entry provide list of your data sources to parse.

Like Reflect always needs a JSON file to run, Remi Collet shared [https://github.com/llaville/php-compat-info/issues/120#issuecomment-46188900] a workaround solution exposed here that allows to run :

```
$ phpreflect analyser:run .
```

plugins
: this entry list all plugins added to the core base code of PHP Reflect.

Don’t forget to add at least this default content, else the `analyser:run` and `analyser:list` commands wouldn’t be available.

analysers
: this entry list all analysers that may be used with the `analyser:run` command.

Don’t miss it, else you could not run the `analyser:run` command.

Chapter 4. Execution

With a default `phpreflect.json` as detailed above, put in the `src/` folder of the Reflect source code, and invoke the following command :

```
$ phpreflect analyser:run .
```

and you should obtain something like this :

```
Data Source Analysed

Directories                                    17
Files                                          53

Structure Analysis
  Namespaces                                   17
  Interfaces                                   10
  Traits                                        0
  Classes                                      51
    Abstract Classes                            4 (7.84%)
    Concrete Classes                           47 (92.16%)
  Methods                                     323
    Scope
      Non-Static Methods                      316 (97.83%)
      Static Methods                            7 (2.17%)
    Visibility
      Public Method                           288 (89.16%)
      Protected Method                         32 (9.91%)
      Private Method                            3 (0.93%)
  Functions                                    6
    Named Functions                            0 (0.00%)
    Anonymous Functions                        6 (100.00%)
  Constants                                    30
    Global Constants                           14 (46.67%)
    Class Constants                            16 (53.33%)
```

Chapter 5. Summary

Let’s review what we’ve done :

- downloaded the latest stable PHAR version.
- prepared your JSON configuration file that is required to run Reflect commands.
- executed your first parse on the Reflect data source.

Chapter 6. Next

Choose your way depending of your skill level.

Read more

- Want to learn more about the command line interpreter (CLI) version, interface that do Reflect an easy tool without to write a line of PHP code, have a look on Part II, “User Guide”
- Want to learn more about Reflect architecture and /or you want to extends it to match your needs, have a look on Part IV, “Developer Guide”
- You are a user of previous version 1.9 that is really different, and want to upgrade to the new major version 2, and keep your old environment still running, have a look on Part III, “Migration Guide”

Part II. User Guide

First visit, you are highly recommended to follow chapters in following order.

1. Installing all necessary Reflect components. See Chapter 7, *Installation*
2. Configuring your project and get ready for your first parsing. See Chapter 8, *The Json Configuration File*
3. Running your first parses with the Command-Line interface. See Chapter 9, *The Command-Line*

All you have to know if you want to upgrade from a previous version 1.x easily.

See Part III, “Migration Guide”

Basic Reflect features does not match your needs. Learn how to extend or change some features/behaviors.

See Part IV, “Developer Guide”

Chapter 7. Installation

Reflect may be installed in several ways, choose your favorite.

Please read the Part III, “Migration Guide” in case you are upgrading from a version 1.x of PHP Reflect.

7.1. Requirements

Before you install PHP Reflect, you will need an operating system with PHP [http://www.php.net] 5.3.0 or later installed,

Reflect requires the date [http://www.php.net/manual/en/book.datetime.php], json [http://www.php.net/manual/en/book.json.php], reflection [http://www.php.net/manual/en/book.reflection.php], tokenizer [http://www.php.net/manual/en/book.tokenizer.php], pcre [http://www.php.net/manual/en/book.pcre.php], and spl [http://www.php.net/manual/en/book.spl.php] extensions. These extensions are usually compiled and enabled by default.

7.2. Composer

Put a file named composer.json at the root of your project, with the content below:

```
{
    "require": {
        "bartlett/php-reflect": "~2.6"
    }
}
```

And ask Composer [http://getcomposer.org/] to install the dependencies:

```
$ php composer.phar install
```

With composer `install` or `create-project` commands, if you want to disable installation of require-dev packages (`doctrine/cache`, `psr/log`, `monolog/monolog`, `bartlett/phpunit-loggertestlistener`), don’t forget to specify the `--no-dev` option.

You can also use Composer to create a new project from an existing Reflect package. This is the equivalent of doing a git clone checkout followed by a composer install of the vendors.

```
$ php composer.phar create-project bartlett/php-reflect /path/to 2.6.3
```

Where `/path/to` is your install directory.

7.3. PHAR

The recommended way for newbies, or just to have a look on features of this library, is to download a PHP Archive that contain all required dependencies of PHP Reflect bundled in a single file.

```
$ wget http://bartlett.laurent-laville.org/get/phpreflect-2.6.3.phar
$ chmod +x phpreflect-2.6.3.phar
$ mv phpreflect-2.6.3.phar /usr/local/bin/phpreflect
$ phpreflect
```

You can also immediately use the PHAR after you have downloaded it.

```
$ wget http://bartlett.laurent-laville.org/get/phpreflect-2.6.3.phar
$ php phpreflect-2.6.3.phar
```

Chapter 8. The Json Configuration File

Reflect always needs a file in JSON [http://json.org/] format to run. It should be found either in the current, `$HOME/.config/`, or `/etc` directory.

By setting the `REFLECT` environment variable it is possible to set the filename of `phpreflect.json` to something else.

E.g: `REFLECT=my-phpreflect.json`

The minimalist JSON file `phpreflect.json` is :

```
{
    "source-providers": [
        {
            "in": ". as current",
            "name": "/\\.(php|inc|phtml)$/"
        }
    ],
    "plugins": [
        {
            "name": "Analyser",
            "class": "Bartlett\\Reflect\\Plugin\\Analyser\\AnalyserPlugin"
        }
    ],
    "analysers" : [
        {
            "name": "Structure",
            "class": "Bartlett\\Reflect\\Analyser\\StructureAnalyser"
        }
    ]
}
```

source-providers
: this entry provide list of your data sources to parse.

plugins
: this entry list all plugins added to the core base code of PHP Reflect.

analysers
: this entry list all analysers that may be used with the `analyser:run` command.

8.1. section Source Providers

There are lot of way to filter your data source. Each rule follow the syntax of Symfony Finder [http://symfony.com/doc/current/components/finder.html] Component.

The **Location** is the only mandatory criteria. It tells the Finder which directory to use for the search.

In a simple directory.

```
    {
        "in": ". as current"
```

```
}
```

If you want to identify a data source easily by a short name, the alias (right of `as`) is compared with the `--alias` option constraint.

Search in several locations.

```
{
    "in": ". as current",
    "in": "src/"
}
```

Use wildcard characters to search in the directories matching a pattern:

```
{
    "in": "src/Bartlett/R*"
}
```

Search directly in archives (phar, zip, tar) with the `phar://` protocol.

```
{
    "in": "phar://path/to/archive.zip"
}
```

Restrict files by name and/or extension.

```
{
    "in": "phar://path/to/archive.zip",
    "name": "*.php"
}
```

Restrict files by size.

```
{
    "in": "phar://path/to/archive.zip",
    "name": "*.php",
    "size": "< 10K"
}
```

Restrict files by last modified dates.

```
{
    "in": ". as current",
    "date": "since yesterday"
}
```

By default, the Finder recursively traverse directories.

Restrict the depth of traversing.

```
{
    "in": ". as current",
    "depth": "< 3"
}
```

Restrict location by only one directory.

```
{
```

```
        "in": ". as current",
        "exclude": "vendor"
    }
```

Restrict location by 1 or more directories.

```
    {
        "in": ". as current",
        "exclude": ["vendor", "tests"]
    }
```

8.2. section Plugins

There are a number of optional plugins you can use along with Reflect to add more capabilities.

The `Analyser` is the only mandatory plugin you should add to parse your data source.

In your `phpreflect.json` configuration file, add in `plugins` section the following entry:

```
    {
        "name": "Analyser",
        "class": "Bartlett\\Reflect\\Plugin\\Analyser\\AnalyserPlugin"
    }
```

The `name` key identify the namespace of optional commands the plugin may provide.

Each `name` key must be unique to avoid conflicts.

The `class` key identify the name of the class that implement the plugin features.

8.2.1. Cache Plugin

Available only since version 2.3.0

In your `phpreflect.json` configuration file, add in `plugins` section the following entry:

```
    {
        "name": "Cache",
        "class": "Bartlett\\Reflect\\Plugin\\Cache\\CachePlugin",
        "options": {
            "adapter": "DoctrineCacheAdapter",
            "backend": {
                "class": "Doctrine\\Common\\Cache\\FilesystemCache",
                "args": [
                    "%{TEMP}/bartlett/cache"
                ]
            }
        }
    }
```

You may use any environment variable that will be replaced, at run-time, by their value. E.g: `TEMP`, `HOME`

Since release 2.3.0, the `HOME` syntax is compatible Linux/Windows.

In previous configuration we used the Doctrine Cache adapter and its File system backend. See the same configuration applied with other SAPI, in Section 17.3, “File cache”

8.2.2. Log Plugin

Available only since version 2.4.0

In your `phpreflect.json` configuration file, add in `plugins` section the following entry:

```
{
    "name": "Log",
    "class": "Bartlett\\Reflect\\Plugin\\Log\\LogPlugin",
    "options": {
        "logger": {
            "class": "YourLogger"
        },
        "conf": []
    }
}
```

Where `YourLogger` identify your class logger (fully qualified. E.g `YourNamespace\YourLogger`).

See the Developer Guide for definition examples of some loggers Section 18.2, “Using your private logger” or Section 18.3, “Using Monolog”

And `conf` entry are custom plugin options to apply.

For example, to suppress logging of `reflect.success` event, and remove contextual data on `reflect.complete` event, modify your `phpreflect.json` configuration file as follow :

```
{
    "name": "Log",
    "class": "Bartlett\\Reflect\\Plugin\\Log\\LogPlugin",
    "options": {
        "logger": {
            "class": "YourLogger"
        },
        "conf": {
            "reflect.success": false,
            "reflect.complete": {
                "context": false
            }
        }
    }
}
```

All configuration options are available, see Section 17.1, “Register Plugin”

8.3. section Analysers

There are a number of optional analysers you can use along with the Reflect Analyser Plugin.

The `Structure` is the default analyser you should add to obtain results when you parse your data source.

In your `phpreflect.json` configuration file, add in `analysers` section the following entry:

```
    {
        "name": "Structure",
        "class": "Bartlett\\Reflect\\Analyser\\StructureAnalyser"
    }
```

The `name` key identify the name you can optionally invoke with the `analyser:run` command.

The two following commands do the same:

Used implicitly the structure analyser (default behavior).

```
$ phpreflect analyser:run .
```

Named explicitly the structure analyser.

```
$ phpreflect analyser:run . structure
```

Each `name` key must be unique to avoid conflicts.

The `class` key identify the name of the class that implement the analyser features.

Chapter 9. The Command-Line

The command-line interface is the easiest way to try and learn the basic Reflect features.

For all users.

9.1. Command-Line Options

Without `plugins` and `analysers` sections in your `phpreflect.json` configuration file, when you invoke the `phpreflect` command, you should obtain the following commands and options :

```
phpReflect version 2.5.0

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v|vv|vvv Increase the verbosity of messages: 1 for normal output, 2 f
  --version        -V Display this application version.
  --ansi              Force ANSI output.
  --no-ansi           Disable ANSI output.
  --no-interaction -n Do not ask any interactive question.
  --profile           Display timing and memory usage information.

Available commands:
  help               Displays help for a command
  list               Lists commands
plugin
  plugin:list        List all plugins installed.
provider
  provider:display   Show source of a file in a data source.
  provider:list      List all data source providers.
  provider:show      Show list of files in a data source.
```

plugin:list — List all plugins configured (and correctly installed) in `plugins` section of your `phpreflect.json` config file.

Without plugins, you will get.

```
$ phpreflect plugin:list
```

```
[Json Configuration]
No plugins detected.
```

With only `Analyser` plugin configured, you will get.

```
$ phpreflect plugin:list
```

```
Plugin Name Plugin Class                                        Events Subscrib
Analyser    Bartlett\Reflect\Plugin\Analyser\AnalyserPlugin reflect.complet
```

provider:list List all data source providers configured in `source-providers` section of your `phpreflect.json` config file.

Result may vary depending of your current directory, but you will get something like.

```
$ phpreflect provider:list
```

```
Source Alias   Files
.      current     46
```

provider:show Show list of files corresponding to the (Symfony) Finder rules defined.

With Reflect source files.

```
$ phpreflect provider:show .
```

Possible alternative.

```
$ phpreflect provider:show --alias current
```

```
Source                                                  Files
.                                                       46
Relative Path Name                                      Date
Bartlett\Reflect\Analyser\AbstractAnalyser.php          2014-02-03T17:25:07
Bartlett\Reflect\Analyser\AnalyserInterface.php         2014-02-03T17:26:50
Bartlett\Reflect\Analyser\StructureAnalyser.php         2014-02-23T17:31:16

<... more lines ...>
```

provider:display Show source code of a file in one of the data source identified.

With `Bartlett\Reflect.php` in the *current* data source.

```
$ phpreflect provider:display . Bartlett\Reflect.php
```

Possible alternative.

```
$ phpreflect provider:display --alias current Bartlett\Reflect.php
```

```
Source
.
Relative Path Name Date Size
Id    Token                          Line  Text
    0 T_OPEN_TAG                        1 <?php
    1 T_DOC_COMMENT                     2 /** * Reflect * Reverse-engineer
    2 T_WHITESPACE                     15
    3 T_NAMESPACE                      17 namespace
    4 T_WHITESPACE                     17
    5 T_STRING                         17 Bartlett
    6 T_SEMICOLON                      17 ;

<... more lines ...>
```

When the `Analyser` plugin is installed, following lines added into `analysers` section

```
    {
        "name": "Analyser",
        "class": "Bartlett\\Reflect\\Plugin\\Analyser\\AnalyserPlugin"
    }
```

you will get two additionnal commands.

analyser:list — List all analysers configured in `analysers` section of your `phpreflect.json` config file.

Without analysers, you will get.

```
$ phpreflect analyser:list
```

```
[Json Configuration]
No analysers detected.
```

With only `Analyser` plugin configured, you will get.

```
$ phpreflect analyser:list
```

```
Analyser Name Analyser Class
Structure     Bartlett\Reflect\Analyser\StructureAnalyser
```

analyser:run — Parse a data source and display results. May wary depending of the data source and analyser used.

With `structure` analyser and the Reflect source code, you will get something like.

```
$ phpreflect analyser:run .
```

Possible alternative.

```
$ phpreflect analyser:run --alias current
```

```
Data Source Analysed

Directories                                          17
Files                                                53

Structure Analysis
  Namespaces                                         17
  Interfaces                                         10
  Traits                                              0
  Classes                                            51
    Abstract Classes                                  4 (7.84%)
    Concrete Classes                                 47 (92.16%)
  Methods                                           323
    Scope
      Non-Static Methods                            316 (97.83%)
      Static Methods                                  7 (2.17%)
    Visibility
      Public Method                                 288 (89.16%)
      Protected Method                               32 (9.91%)
      Private Method                                  3 (0.93%)
  Functions                                           6
    Named Functions                                   0 (0.00%)
    Anonymous Functions                               6 (100.00%)
  Constants                                          30
    Global Constants                                 14 (46.67%)
    Class Constants                                  16 (53.33%)
```

When the `plantUML` plugin is installed, following lines added into `plugins` section

```
    {
        "name": "PlantUML",
        "class": "Bartlett\\Reflect\\Plugin\\PlantUML\\PlantUMLPlugin"
    }
```

you will get one additionnal command.

plantUML:run Parse a data source and draw corresponding UML diagrams (package or class) with the PlantUML [http://plantuml.sourceforge.net/] syntax.

Build a package diagram corresponding to `Bartlett\Reflect\Model`.

```
$ phpreflect plantUML:run --package=Bartlett\Reflect\Model .
```

Possible Alternative.

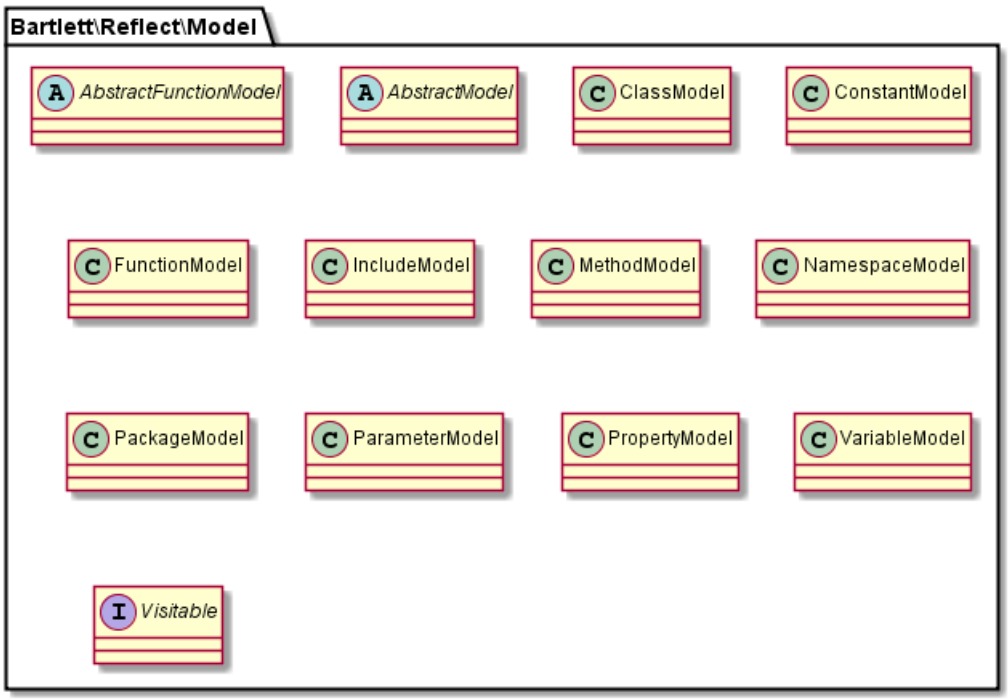
```
$ phpreflect plantUML:run --package=Bartlett\Reflect\Model --alias current
```

```
PlantUML package diagram
```

It's now time to produce PNG images. Put previous PlantUML code syntax into a text file (e.g named `packageDiagram.plantuml`), and run the following command.

```
$ java -jar plantuml.jar packageDiagram.plantuml
```

And you should obtain a PNG image like this one



Build a class diagram corresponding to `Bartlett\Reflect\Builder`.

```
$ phpreflect plantUML:run --class=Bartlett\Reflect\Builder .
```

Possible Alternative.

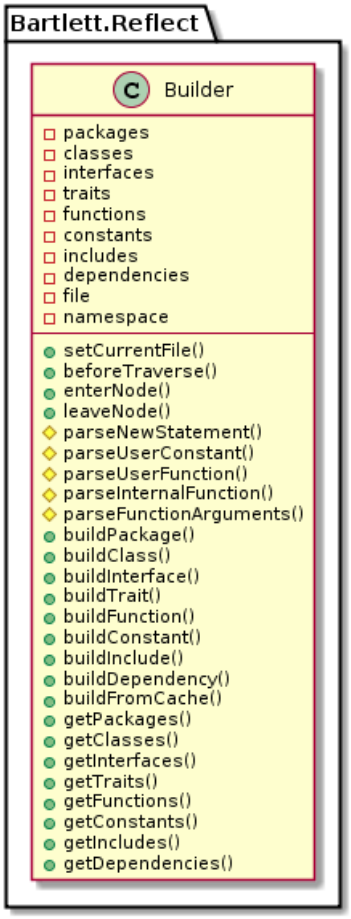
```
$ phpreflect plantUML:run --class=Bartlett\Reflect\Builder --alias current
```

```
PlantUML class diagram
```

It’s now time to produce PNG images. Put previous PlantUML code syntax into a text file (e.g named `classDiagram.plantuml`), and run the following command.

```
$ java -jar plantuml.jar classDiagram.plantuml
```

And you should obtain a PNG image like this one



Chapter 10. Summary

Let’s review what we’ve learned about the command-line interface :

- It’s a Symfony Console Component [http://symfony.com/doc/current/components/console/index.html] that can be extended to infinite via plugins and analysers.
- Default analyser produced results such as PHPLoc [https://github.com/sebastianbergmann/phploc] by Sebastian Bergmann.
- May generate class and package UML diagrams with PlantUML [http://plantuml.sourceforge.net/].

Part III. Migration Guide

Because the version 2 is a full API rewrites, and used namespaces, your old code cannot migrate without a little change.

We will try to explain how to do in few steps.

Chapter 11. Containers

Version 1.x used customizable containers [http://php5.laurent-laville.org/reflect/manual/1.9/en/configure.html#containers] feature to store parsing results.

For example, in **version 1.x** when we wanted to retrieve user functions, we could either do :

Example 11.1. With standard container

```
<?php
require_once 'Bartlett/PHP/Reflect/Autoload.php';

$source = '/path/to/source_file.php';

$options = array();

$reflect = new PHP_Reflect($options);
$reflect->scan($source);

$functions = $reflect->getFunctions();
// OR
$functions = $reflect['functions'];
```

Example 11.2. With a non standard container

```
<?php
require_once 'Bartlett/PHP/Reflect/Autoload.php';

$source = '/path/to/source_file.php';

$options = array('containers' => array('function' => 'userFunctions');

$reflect = new PHP_Reflect($options);
$reflect->scan($source);

$functions = $reflect->getUserFunctions();
// OR
$functions = $reflect['userFunctions'];
```

In **version 2.x**, we have collections of data models that we can enumerate and exploit.

See API in developer's guide

- Namespaces collection. See Section 15.4.1, "Packages or Namespaces"
- Classes collection. See Section 15.4.3, "Classes"
- Interfaces collection. See Section 15.4.4, "Interfaces"
- Traits collection. See Section 15.4.5, "Traits"
- Functions collection. See Section 15.4.8, "Functions"
- Constants collection. See Section 15.4.9, "Constants"
- Includes collection. See Section 15.4.10, "Includes"
- Dependencies collection. See Section 15.4.11, "Dependencies"

Example 11.3. Enumerate each user functions

```
<?php
require_once 'vendor/autoload.php';

use Bartlett\Reflect;
use Bartlett\Reflect\ProviderManager;
use Bartlett\Reflect\Provider\SymfonyFinderProvider;

use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()
    ->name('source_file.php')
    ->in('/path/to/');

// Identify Data Source
$pm = new ProviderManager;
$pm->set('Sample', new SymfonyFinderProvider($finder));

$reflect = new Reflect;
$reflect->setProviderManager($pm);
$reflect->parse();

// Exploit results
foreach ($reflect->getPackages() as $package) {
    $functions = $package->getFunctions();
}
```

Chapter 12. Properties

Version 1.x may provide a variable properties [http://php5.laurent-laville.org/reflect/manual/1.9/en/configure.html#properties] list. Version 2.x provides all properties anytime. It’s up to you to decide to use them or not.

For example, in **version 1.x** when we wanted to retrieve only keywords and signature of each class methods of a data source.

Example 12.1. Configure interface, class and method properties

```
<?php
require_once 'Bartlett/PHP/Reflect/Autoload.php';

$source = '/path/to/PEAR-1.9.2/PEAR.php';

$options = array(
    'properties' => array(
        'interface' => array(
            'parent', 'methods'
        ),
        'class' => array(
            'parent', 'methods', 'interfaces', 'package'
        ),
        'function' => array(
            'signature'
        ),
    )
);
$reflect = new PHP_Reflect($options);
$reflect->scan($source);

$classes = $reflect->getClasses();

print_r($classes['\\']['PEAR_Error']['methods']);
```

Script output.

```
Array
(
    [PEAR_Error] => Array
        (
            [signature] => PEAR_Error($message = 'unknown error', $code = null,
                           $mode = null, $options = null, $userinfo = null)
        )

    [getMode] => Array
        (
            [signature] => getMode()
        )

    [getCallback] => Array
        (
            [signature] => getCallback()
        )

    [getMessage] => Array
```

```
        (
            [signature] => getMessage()
        )

    [getCode] => Array
        (
            [signature] => getCode()
        )

    [getType] => Array
        (
            [signature] => getType()
        )

    [getUserInfo] => Array
        (
            [signature] => getUserInfo()
        )

    [getDebugInfo] => Array
        (
            [signature] => getDebugInfo()
        )

    [getBacktrace] => Array
        (
            [signature] => getBacktrace($frame = null)
        )

    [addUserInfo] => Array
        (
            [signature] => addUserInfo($info)
        )

    [__toString] => Array
        (
            [signature] => __toString()
        )

    [toString] => Array
        (
            [signature] => toString()
        )
)
```

In **version 2.x**, when we did the same.

Example 12.2. Properties on demand

```
<?php
require_once 'vendor/autoload.php';

use Bartlett\Reflect;
use Bartlett\Reflect\ProviderManager;
use Bartlett\Reflect\Provider\SymfonyFinderProvider;

use Symfony\Component\Finder\Finder;

$finder = new Finder();
```

```
$finder->files()
    ->name('PEAR.php')
    ->in('/path/to/PEAR-1.9.2/');

// Identify Data Source
$pm = new ProviderManager;
$pm->set('PEAR192', new SymfonyFinderProvider($finder));

$reflect = new Reflect;
$reflect->setProviderManager($pm);
$reflect->parse();

// Exploit results
$out = array();

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {

        if ($class->getShortName() !== 'PEAR_Error') {
            continue;
        }

        foreach ($class->getMethods() as $method) {

            if ($method->isPrivate()) {
                $visibility = 'private';
            } elseif ($method->isProtected()) {
                $visibility = 'protected';
            } else {
                $visibility = 'public';
            }

            $name = $method->getShortName();

            $parameters = $method->getParameters();
            $args       = array();

            foreach ($parameters as $parameter) {

                $args[] = sprintf(
                    '%s%s%s',
                    $parameter->isPassedByReference() ? '&' : '',
                    '$' . $parameter->getName(),
                    $parameter->isDefaultValueAvailable() ? ' = ' . $parameter->getDefau
                );
            }

            $out[$name] = array(
                'signature' => sprintf('%s %s(%s)', $visibility, $name, implode(',', $arg
            );
        }
    }
}
print_r($out);
```

Script output.

```
Array
(
```

```
    [PEAR_Error] => Array
        (
            [signature] => public PEAR_Error($message = 'unknown error',$code = null,$mo
        )

    [getMode] => Array
        (
            [signature] => public getMode()
        )

    [getCallback] => Array
        (
            [signature] => public getCallback()
        )

    [getMessage] => Array
        (
            [signature] => public getMessage()
        )

    [getCode] => Array
        (
            [signature] => public getCode()
        )

    [getType] => Array
        (
            [signature] => public getType()
        )

    [getUserInfo] => Array
        (
            [signature] => public getUserInfo()
        )

    [getDebugInfo] => Array
        (
            [signature] => public getDebugInfo()
        )

    [getBacktrace] => Array
        (
            [signature] => public getBacktrace($frame = null)
        )

    [addUserInfo] => Array
        (
            [signature] => public addUserInfo($info)
        )

    [__toString] => Array
        (
            [signature] => public __toString()
        )

    [toString] => Array
        (
            [signature] => public toString()
        )
```

```
)
```

Chapter 13. Summary

Let’s review what we’ve did :

- Compared **Containers** configuration solutions, and how to do it with both versions **1.x** and **2.x**
- Compared **Properties** configuration solutions, and how to do it with both versions **1.x** and **2.x**
- Used some methods of the new **API 2.x**, to enumerate and exploit parsing results.

Part IV. Developer Guide

 **API**

Reflect comes with a complete reflection API, almost equivalent to PHP5 reflection [http://www.php.net/manual/en/book.reflection.php].

See Chapter 15, *API*

 **Plugins**

Reflect uses a Symfony EventDispatcher [http://symfony.com/doc/current/components/event_dispatcher/index.html] Component to allow you to easily extend the features list.

See Chapter 16, *Plugins*

- Use cache plugin to speed up future data source parsing.

 **Analysers**

Reflect uses analysers that implements the Visitor [http://en.wikipedia.org/wiki/Visitor_pattern] pattern in a simple and effective way to make the render of your results truly customizable.

See Chapter 19, *Analysers*

Chapter 14. API compared

14.1. Class Reflection

PHP5	Reflect	Description
Class reports information about a class http://www.php.net/manual/en/class.reflectionclass.php `Bartlett\Reflect\Model\ClassModel`		
☑	☑	**__construct** - Constructs a ReflectionClass
☑	☑	**__toString** - Returns the string representation of the ReflectionClass object
☑	☐	**export** - Exports a class
☑	☑	**getConstant** - Gets defined constant
☑	☑	**getConstants** - Gets constants
☑	☐	**getConstructor** - Gets the constructor of the class
☑	☐	**getDefaultProperties** - Gets default properties
☑	☑	**getDocComment** - Gets doc comments
☑	☑	**getEndLine** - Gets end line
☑	☑	**getExtension** - Gets a ReflectionExtension object for the extension which defined the class
☑	☑	**getExtensionName** - Gets the name of

PHP5	Reflect	Description
		the extension which defined the class
☑	☑	**getFileName** - Gets the filename of the file in which the class has been defined
☑	☑	**getInterfaceNames** - Gets the interface names
☑	☑	**getInterfaces** - Gets the interfaces
☑	☑	**getMethod** - Gets a ReflectionMethod for a class method
☑	☑	**getMethods** - Gets an array of methods
☑	☐	**getModifiers** - Gets modifiers
☑	☑	**getName** - Gets class name
☑	☑	**getNamespaceName** - Gets namespace name
☑	☑	**getParentClass** - Gets parent class
☑	☑	**getProperties** - Gets properties
☑	☑	**getProperty** - Gets a ReflectionProperty for a class's property
☑	☑	**getShortName** - Gets short name
☑	☑	**getStartLine** - Gets starting line number
☑	☑	**getStaticProperties** - Gets static properties

PHP5	Reflect	Description
☑	☑	**getStaticPropertyValue** - Gets static property value
☑	☐	**getTraitAliases** - Returns an array of trait aliases
☑	☐	**getTraitNames** - Returns an array of names of traits used by this class
☑	☐	**getTraits** - Returns an array of traits used by this class
☑	☑	**hasConstant** - Checks if constant is defined
☑	☑	**hasMethod** - Checks if method is defined
☑	☑	**hasProperty** - Checks if property is defined
☑	☐	**implementsInterface** - Implements interface
☑	☑	**inNamespace** - Checks if class in namespace
☑	☑	**isAbstract** - Checks if class is abstract
☑	☑	**isCloneable** - Returns whether this class is cloneable
☑	☑	**isFinal** - Checks if class is final
☑	☐	**isInstance** - Checks class for instance
☑	☑	**isInstantiable** - Checks if the class is instantiable

PHP5	Reflect	Description
☑	☑	**isInterface** - Checks if the class is an interface
☑	☐	**isInternal** - Checks if class is defined internally by an extension, or the core
☑	☑	**isIterateable** - Checks if iterateable
☑	☑	**isSubclassOf** - Checks if a subclass
☑	☑	**isTrait** - Returns whether this is a trait
☑	☑	**isUserDefined** - Checks if user defined
☑	☐	**newInstance** - Creates a new class instance from given arguments
☑	☐	**newInstanceArgs** - Creates a new class instance from given arguments
☑	☐	**newInstanceWithoutConstr** - Creates a new class instance without invoking the constructor
☑	☐	**setStaticPropertyValue** - Sets static property value

14.2. Constant Reflection

Does not exist in PHP5 Reflection API

PHP5	Reflect	Description
Class reports information about a constant `Bartlett\Reflect\Model\ConstantModel`		

PHP5	Reflect	Description
☐	☑	**__construct** - Constructs a Reflection Constant
☐	☑	**__toString** - Returns the string representation of the Reflection Constant object
☐	☑	**getDocComment** - Gets doc comments
☐	☑	**getExtension** - Gets a ReflectionExtension object for the extension which defined the constant
☐	☑	**getExtensionName** - Gets the name of the extension which defined the constant
☐	☑	**getFileName** - Gets the filename of the file in which the constant has been defined
☐	☑	**getName** - Gets constant name
☐	☑	**getNamespaceName** - Gets namespace name
☐	☑	**getShortName** - Gets short name
☐	☑	**getValue** - Gets value
☐	☑	**inNamespace** - Checks if in namespace
☐	☑	**isInternal** - Checks if constant is defined internally by an extension, or the core

PHP5	Reflect	Description
☐	☑	**isMagic** - Checks whether it's a magic constant

14.3. Function Reflection

PHP5	Reflect	Description
Class reports information about a function http://www.php.net/manual/en/class.reflectionfunction.php `Bartlett\Reflect\Model\FunctionModel`		
☑	☑	**__construct** - Constructs a ReflectionFunction
☑	☑	**__toString** - Returns the string representation of the ReflectionFunction object
☑	☐	**export** - Exports a function
☑	☐	**getClosure** - Returns a dynamically created closure for the function
☑	☐	**invoke** - Invokes function
☑	☐	**invokeArgs** - Invokes function with args
☑	☐	**isDisabled** - Checks if function is disabled

14.4. Function Abstract Reflection

PHP5	Reflect	Description
A parent class to ReflectionFunction http://www.php.net/manual/en/class.reflectionfunctionabstract.php `Bartlett\Reflect\Model\AbstractFunctionModel`		
☑	☐	**__clone** - Clones function

PHP5	Reflect	Description
☑	☐	**getClosureScopeClass** - Returns the scope associated to the closure
☑	☐	**getClosureThis** - Returns this pointer bound to closure
☑	☑	**getDocComment** - Gets doc comments
☑	☑	**getEndLine** - Gets end line
☑	☑	**getExtension** - Gets a ReflectionExtension object for the extension which defined the function
☑	☑	**getExtensionName** - Gets the name of the extension which defined the function
☑	☑	**getFileName** - Gets the filename of the file in which the function has been defined
☑	☑	**getName** - Gets function name
☑	☑	**getNamespaceName** - Gets namespace name
☑	☑	**getNumberOfParameters** - Gets number of parameters
☑	☑	**getNumberOfRequiredPara** - Gets number of required parameters
☑	☑	**getParameters** - Gets parameters

PHP5	Reflect	Description
☑	☑	**getShortName** - Gets function short name
☑	☑	**getStartLine** - Gets starting line number
☑	☐	**getStaticVariables** - Gets static variables
☑	☑	**inNamespace** - Checks if function in namespace
☑	☑	**isClosure** - Checks if closure
☑	☐	**isDeprecated** - Checks if function deprecated
☑	☐	**isGenerator** - Returns whether this function is a generator
☑	☑	**isInternal** - Checks if function is defined internally by an extension, or the core
☑	☐	**isUserDefined** - Checks if user defined
☑	☐	**returnsReference** - Checks if returns reference

14.5. Method Reflection

PHP5	Reflect	Description
Class reports information about a method http://www.php.net/manual/en/class.reflectionmethod.php `Bartlett\Reflect\Model\MethodModel`		
☑	☑	**__construct** - Constructs a ReflectionMethod

PHP5	Reflect	Description
☑	☑	**__toString** - Returns the string representation of the ReflectionMethod object
☑	☐	**export** - Exports a method
☑	☐	**getClosure** - Returns a dynamically created closure for the method
☑	☐	**getDeclaringClass** - Gets declaring class for the reflected method
☑	☐	**getModifiers** - Gets the method modifiers
☑	☐	**getPrototype** - Gets the method prototype
☑	☐	**invoke** - Invokes method
☑	☐	**invokeArgs** - Invokes method with args
☑	☑	**isAbstract** - Checks if method is abstract
☑	☑	**isConstructor** - Checks if method is a constructor
☑	☑	**isDestructor** - Checks if method is a destructor
☑	☑	**isFinal** - Checks if method is final
☑	☑	**isPrivate** - Checks if method is private

PHP5	Reflect	Description
☑	☑	**isProtected** - Checks if method is protected
☑	☑	**isPublic** - Checks if method is public
☑	☑	**isStatic** - Checks if method is static
☑	☐	**setAccessible** - Set method accessibility

14.6. Parameter Reflection

PHP5	Reflect	Description
Class reports information about a parameter http://www.php.net/manual/en/class.reflectionparameter.php `Bartlett\Reflect\Model\ParameterModel`		
☑	☑	**__construct** - Constructs a ReflectionParameter
☑	☑	**__toString** - Returns the string representation of the ReflectionParameter object
☑	☐	**__clone** - Clones parameter
☑	☑	**allowsNull** - Checks if null is allowed
☑	☐	**canBePassedByValue** - Returns whether this parameter can be passed by value
☑	☐	**export** - Exports a parameter
☑	☐	**getClass** - Gets class
☐	☑	**getTypeHint** - Gets the type of the

PHP5	Reflect	Description
		parameter (callable, array, class name, or none)
☑	☐	**getDeclaringClass** - Gets declaring class for the reflected parameter
☑	☐	**getDeclaringFunction** - Gets declaring function for the reflected parameter
☑	☑	**getDefaultValue** - Gets default parameter value
☑	☐	**getDefaultValueConstantNa** - Returns the default value's constant name if default value is constant or null
☑	☑	**getName** - Gets parameter name
☑	☑	**getPosition** - Gets parameter position
☑	☑	**isArray** - Checks if parameter expects an array
☑	☐	**isCallable** - Returns whether parameter MUST be callable
☑	☑	**isDefaultValueAvailable** - Checks if a default value is available
☑	☐	**isDefaultValueConstant** - Returns whether the default value of this parameter is constant
☑	☑	**isOptional** - Checks if the parameter is optional
☑	☑	**isPassedByReference** - Checks if the

PHP5	Reflect	Description
		parameter is passed in by reference

14.7. Property Reflection

PHP5	Reflect	Description
Class reports information about classes properties http://www.php.net/manual/en/class.reflectionproperty.php `Bartlett\Reflect\Model\PropertyModel`		
☑	☑	**__construct** - Constructs a ReflectionProperty
☑	☑	**__toString** - Returns the string representation of the ReflectionProperty object
☑	☐	**__clone** - Clones property
☑	☐	**export** - Exports a property
☐	☑	**getClassName** - Gets class name of the reflected property
☑	☐	**getDeclaringClass** - Gets declaring class for the reflected property
☑	☑	**getDocComment** - Gets doc comments from a property
☑	☐	**getModifiers** - Gets modifiers
☑	☑	**getName** - Gets property name
☑	☑	**getValue** - Gets property value
☑	☑	**isDefault** - Checks if default value

PHP5	Reflect	Description
☑	☑	**isPrivate** - Checks if property is private
☑	☑	**isProtected** - Checks if property is protected
☑	☑	**isPublic** - Checks if property is public
☑	☑	**isStatic** - Checks if property is static
☑	☐	**setAccessible** - Set property accessibility
☑	☐	**setValue** - Set property value

Chapter 15. API

15.1. Data Source Identification

Identify the Data Source with the Symfony Finder [http://symfony.com/doc/current/components/finder.html] Component.

Now, and for the following chapters, we will not mention how you load the classes. Depending of the install strategy you've adopted, Composer or other, don't forget to load your autoloader.

Reflect offers a data source provider mechanism. You may either use the basic Symfony Finder, what we will do next, or use your own.

15.1.1. Basic Provider with Symfony Finder

```
<?php

use Bartlett\Reflect\Provider\SymfonyFinderProvider;
use Symfony\Component\Finder\Finder;

$dirs = dirname(__DIR__) . '/sources';

$finder = new Finder();
$finder->files()
    ->name('*.php')
    ->in($dirs);

$provider = new SymfonyFinderProvider($finder);
```

At this step, we have created a data source provider that is allowed to retrieve each element to parse.

Reflect need to know it. We attach then the previous provider instance to a **Provider Manager**, with a label (e.g: `Single`) to identify it easily.

Reflect Provider Manager with a unique data source.

```
<?php

use Bartlett\Reflect\ProviderManager;

$pm = new ProviderManager;
$pm->set('Single', $provider);
```

A **Provider Manager** may provide one or more data source identifications. Equivalent to the `source-providers` section of the `phpreflect.json` configuration file for the command-line interface.

Reflect Provider Manager with multiple data sources.

```
<?php

use Bartlett\Reflect\ProviderManager;
```

```
use Bartlett\Reflect\Provider\SymfonyFinderProvider;
use Symfony\Component\Finder\Finder;

$pm = new ProviderManager;

// -- source 1
$source1 = dirname(__DIR__) . '/sources/';

$finder1 = new Finder();
$finder1->files()
    ->name('sample1.php')
    ->in($source1);

$pm->set('Sample', new SymfonyFinderProvider($finder1));

// -- source 2
$pharFile = dirname(__DIR__) . '/sources/pirus.phar';
$source2  = 'phar://' . $pharFile;

$finder2 = new Finder();
$finder2->files()
    ->path('/Pirus/')
    ->name('*.php')
    ->in($source2);

$pm->set('Pirus', new SymfonyFinderProvider($finder2));
```

On this example Reflect is able to parse contents of two data sources: `Sample` and `Pirus`, all at once (default behavior) or individually.

15.2. Parse elements of the provider

We reuse the provider manager instance (`$pm`) seen above (unique data source named `Single`). Then we ask Reflect to parse its full contents.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;
$reflect->setProviderManager($pm);
$reflect->parse();
```

In case of multiple data sources, when you want to parse it individually rather than fully, use the following statements.

Parse only Data Source named `Pirus`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;
$reflect->setProviderManager($pm);
$reflect->parse(array('Pirus'));
```

`Pirus` is the data source label used on `$pm#set()` statement.

You have identified data sources and parsed its full contents. Now you are ready to handle the results.

15.3. Study case

We are suppose in the following, study the source code below, script named `StudyCase1.php` :

```
<?php
include '/path/to/test1.php';
include_once 'test2.php';
require 'test3.php';
// test four
require_once
    'test4.php';

class Foo implements SplObserver
{
    const FOO = 'default FOO value';

    public function __construct()
    {
    }

    public function update(SplSubject $subject) {
    }
}

define('SOURCE1_ROOT', __DIR__);

function singleFunction( Array $someparam, stdClass $somethingelse, $lastone = NULL )
{
}
```

To explore and exploit results, we need first to parse the data source. You should be able to do it.

Here are the solution how to parse the data source seen above :

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\ProviderManager;
use Bartlett\Reflect\Provider\SymfonyFinderProvider;
use Symfony\Component\Finder\Finder;

$dirs = dirname(__DIR__) . '/sources';

$finder = new Finder();
$finder->files()
    ->name('StudyCase1.php')
    ->in($dirs);

$provider = new SymfonyFinderProvider($finder);

$pm = new ProviderManager;
$pm->set('StudyCase1', $provider);

$reflect = new Reflect;
$reflect->setProviderManager($pm);
$reflect->parse();
```

15.4. Enumerate each elements

15.4.1. Packages or Namespaces

Your first collection returned could be the list of packages defined in your source code, if any.

Packages or Namespaces are used to organize software elements to avoid conflicts.

Indifferently, we will use the term package or namespace as an alias of the other. Main reason is that PHP5 provides only namespace and not package element as Java.

Reflect returns a list of namespaces/packages with the `getPackages()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

$packages = $reflect->getPackages();
```

This list (`$packages`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit namespaces.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    echo $package->getName(), PHP_EOL;
}
```

15.4.2. Uses

Your last collection returned could be the list of use statements defined in a package, if any.

Reflect returns a list of use statements with the `getUses()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $uses = $package->getUses();
}
```

This list (`$uses`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit uses.

```
<?php
```

```
use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getUses() as $use) {
        printf('%s with alias %s%s', $use->getName(), $use->getShortName(), PHP_EOL);
    }
}
```

15.4.3. Classes

Your second collection returned could be the list of classes defined in a package, if any.

Reflect returns a list of classes with the `getClasses()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $classes = $package->getClasses();
}
```

This list (`$classes`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit classes.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        echo $class->getName(), PHP_EOL;
    }
}
```

15.4.4. Interfaces

Your next collection returned could be the list of interfaces defined in a package, if any.

Reflect returns a list of interfaces with the `getInterfaces()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $interfaces = $package->getInterfaces();
}
```

This list (`$interfaces`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit interfaces.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getInterfaces() as $interface) {
        echo $interface->getName(), PHP_EOL;
    }
}
```

15.4.5. Traits

Your next collection returned could be the list of traits defined in a package, if any.

Reflect returns a list of traits with the `getTraits()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $traits = $package->getTraits();
}
```

This list (`$traits`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit traits.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getTraits() as $trait) {
        echo $trait->getName(), PHP_EOL;
    }
}
```

15.4.6. Properties

Your next collection returned could be the list of class properties defined in a package, if any.

Reflect returns a list of class properties with the `getProperties()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        printf( 'Processing class "%s" ...' . PHP_EOL, $class->getName() );
        $properties = array();
```

```
        foreach ($class->getProperties() as $property) {
            $properties[] = $property->getName();
        }
        printf( 'Properties are : %s' . PHP_EOL, print_r($properties, true) );
    }
}
```

15.4.7. Parameters

Your next collection returned could be the list of function parameters defined, if any.

Reflect returns a list of function parameters with the `getParameters()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        printf( 'Processing class "%s" ...' . PHP_EOL, $class->getName() );

        foreach ($class->getMethods() as $method) {
            $parameters = array();

            foreach ($method->getParameters() as $parameter) {
                $parameters[] = $parameter->getName();
            }
            printf( 'Parameters are : %s' . PHP_EOL, print_r($parameters, true) );
        }
    }
}
```

15.4.8. Functions

Your next collection returned could be the list of functions defined in a package, if any.

Reflect returns a list of user functions with the `getFunctions()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $functions = $package->getFunctions();
}
```

This list (`$functions`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit functions.

```
<?php
```

```
use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getFunctions() as $function) {
        echo $function->getName(), PHP_EOL;
    }
}
```

15.4.9. Constants

Your next collection returned could be the list of constants defined in a package, if any.

Reflect returns a list of constants with the `getConstants()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $constants = $package->getConstants();
}
```

This list (`$constants`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit constants.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getConstants() as $constant) {
        echo $constant->getName(), PHP_EOL;
    }
}
```

15.4.10. Includes

Your next collection returned could be the list of includes defined in a package, if any.

Reflect returns a list of includes with the `getIncludes()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $includes = $package->getIncludes();
}
```

This list (`$includes`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit includes.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getIncludes() as $include) {
        echo $include->getFilePath(), PHP_EOL;
    }
}
```

15.4.11. Dependencies

Your last collection returned could be the list of dependencies defined in a package, if any.

Reflect returns a list of dependencies with the `getDependencies()` method.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $dependencies = $package->getDependencies();
}
```

This list (`$dependencies`) is an iterator, that can be traversed by a simple foreach loop.

Easy exploit dependencies.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getDependencies() as $dep) {
        echo $dep->getName(), PHP_EOL;
    }
}
```

15.5. Exploit each elements

15.5.1. Packages or Namespaces

The `Bartlett\Reflect\Model\PackageModel` class reports information about a package/namespace.

Reflect returns a list of packages with the `getPackages()` method, and each of these elements can be exploited with this `PackageModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;
```

```
$packages = $reflect->getPackages();
```

This list (`$packages`) is an iterator, that can be traversed by a simple foreach loop. Each `$package` element returned is an instance of `PackageModel`.

Gets an array of element counters.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    $pkgName = $package->getName();
    printf( 'Processing package "%s" ...' . PHP_EOL, $pkgName );

    $counters = array(
        'classes'    => count($package->getClasses()),
        'interfaces' => count($package->getInterfaces()),
        'traits'     => count($package->getTraits()),
        'functions'  => count($package->getFunctions()),
        'constants'  => count($package->getConstants()),
        'includes'   => count($package->getIncludes()),
    );

    printf( 'Metrics : %s' . PHP_EOL, print_r($counters, true) );
}
```

And lot more. See PackageModel Reference to learn all features and behaviors.

`PackageModel` Reference is not yet available.

15.5.2. Uses

The `Bartlett\Reflect\Model\UseModel` class reports information about a use to import a standard namespace or just a constant or function.

Reflect returns a list of use statements with the `getUses()` method, and each of these elements can be exploited with this `UseModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $uses = $package->getUses();
}
```

This list (`$uses`) is an iterator, that can be traversed by a simple foreach loop.

Gets an array of use statements.

```
<?php
```

```
use Bartlett\Reflect;

$uses = array();

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getUses() as $use) {
        $uses[] = array('name' => $use->getName(), 'alias' => $use->getShortName());
    }
}
print_r($uses);
```

And lot more. See UseModel Reference to learn all features and behaviors.

`UseModel` Reference is not yet available.

15.5.3. Classes

The `Bartlett\Reflect\Model\ClassModel` class reports information about a class, an interface or a trait.

Reflect returns a list of classes with the `getClasses()` method, and each of these elements can be exploited with this `ClassModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $classes = $package->getClasses();
}
```

This list (`$classes`) is an iterator, that can be traversed by a simple foreach loop. Each `$class` element returned is an instance of `ClassModel`.

Gets an array of methods for the class.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        printf( 'Processing class "%s" ...' . PHP_EOL, $class->getName() );
        $methods = array();

        foreach ($class->getMethods() as $method) {
            $methods[] = $method->getShortName();
        }
        printf( 'Methods are : %s' . PHP_EOL, print_r($methods, true) );
    }
}
```

Gets an array of constants for the class.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        printf( 'Processing class "%s" ...' . PHP_EOL, $class->getName() );
        $constants = array();

        foreach ($class->getConstants() as $constant) {
            $constants[ $constant->getShortName() ] = $constant->getValue();
        }
        printf( 'Constants are : %s' . PHP_EOL, print_r($constants, true) );
    }
}
```

Gets an array of properties for the class.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        printf( 'Processing class "%s" ...' . PHP_EOL, $class->getName() );
        $properties = array();

        foreach ($class->getProperties() as $property) {
            $properties[] = $property->getName();
        }
        printf( 'Properties are : %s' . PHP_EOL, print_r($properties, true) );
    }
}
```

And lot more. See ClassModel Reference to learn all features and behaviors.

`ClassModel` Reference is not yet available.

15.5.4. Interfaces

The `Bartlett\Reflect\Model\ClassModel` class reports information about a class, an interface or a trait.

Reflect returns a list of interfaces with the `getInterfaces()` method, and each of these elements can be exploited with this `ClassModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $interfaces = $package->getInterfaces();
}
```

This list (`$interfaces`) is an iterator, that can be traversed by a simple foreach loop. Each `$interface` element returned is an instance of `ClassModel`.

Gets an array of methods for the interface.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getInterfaces() as $interface) {
        printf( 'Processing interface "%s" ...' . PHP_EOL, $interface->getName() );
        $methods = array();

        foreach ($interface->getMethods() as $method) {
            $methods[] = $method->getShortName();
        }
        printf( 'Methods are : %s' . PHP_EOL, print_r($methods, true) );
    }
}
```

And lot more. See ClassModel Reference to learn all features and behaviors.

`ClassModel` Reference is not yet available.

15.5.5. Traits

The `Bartlett\Reflect\Model\ClassModel` class reports information about a class, an interface or a trait.

Reflect returns a list of traits with the `getTraits()` method, and each of these elements can be exploited with this `ClassModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $traits = $package->getTraits();
}
```

This list (`$traits`) is an iterator, that can be traversed by a simple foreach loop. Each `$interface` element returned is an instance of `ClassModel`.

Gets an array of methods for the trait.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getTraits() as $trait) {
```

```
        printf( 'Processing trait "%s" ...' . PHP_EOL, $trait->getName() );
        $methods = array();

        foreach ($trait->getMethods() as $method) {
            $methods[] = $method->getShortName();
        }
        printf( 'Methods are : %s' . PHP_EOL, print_r($methods, true) );
    }
}
```

And lot more. See ClassModel Reference to learn all features and behaviors.

`ClassModel` Reference is not yet available.

15.5.6. Properties

The `Bartlett\Reflect\Model\PropertyModel` class reports information about a class property.

Reflect returns a list of class properties with the `getProperties()` method, and each of these elements can be exploited with this `PropertyModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        $properties = $class->getProperties();
    }
}
```

This list (`$properties`) is an iterator, that can be traversed by a simple foreach loop. Each `$properties` element returned is an instance of `PropertyModel`.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        printf( 'Processing class "%s" ...' . PHP_EOL, $class->getName() );
        $properties = array();

        foreach ($class->getProperties() as $property) {
            $properties[] = $property->getName();
        }
        printf( 'Properties are : %s' . PHP_EOL, print_r($properties, true) );
    }
}
```

`PropertyModel` Reference is not yet available.

15.5.7. Parameters

The `Bartlett\Reflect\Model\ParameterModel` class reports information about a function parameter.

Reflect returns a list of function parameters with the `getParameters()` method, and each of these elements can be exploited with this `ParameterModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        foreach ($class->getMethods() as $method) {
            $parameters = $method->getParameters();
        }
    }
}
```

This list (`$parameters`) is an iterator, that can be traversed by a simple foreach loop. Each `$parameters` element returned is an instance of `ParameterModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getClasses() as $class) {
        printf( 'Processing class "%s" ...' . PHP_EOL, $class->getName() );

        foreach ($class->getMethods() as $method) {
            $parameters = array();

            foreach ($method->getParameters() as $parameter) {
                $parameters[] = $parameter->getName();
            }
            printf( 'Parameters are : %s' . PHP_EOL, print_r($parameters, true) );
        }
    }
}
```

`ParameterModel` Reference is not yet available.

15.5.8. Functions

The `Bartlett\Reflect\Model\FunctionModel` class reports information about a function.

Reflect returns a list of user functions with the `getFunctions()` method, and each of these elements can be exploited with this `FunctionModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $functions = $package->getFunctions();
}
```

This list (`$functions`) is an iterator, that can be traversed by a simple foreach loop. Each `$function` element returned is an instance of `FunctionModel`.

Gets an array of functions for the data source.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getFunctions() as $function) {
        printf( 'Processing function "%s" ...' . PHP_EOL, $function->getName() );
        $parameters = array();

        foreach ($function->getParameters() as $parameter) {

            $parameters[] = array(
                'position' => $parameter->getPosition(),
                'optional' => $parameter->isOptional() ? 'YES' : 'NO',
                'name'     => $parameter->getName(),
            );
        }
        printf( 'Parameters are : %s' . PHP_EOL, print_r($parameters, true) );
    }
}
```

And lot more. See FunctionModel Reference to learn all features and behaviors.

`FunctionModel` Reference is not yet available.

15.5.9. Constants

The `Bartlett\Reflect\Model\ConstantModel` class reports information about a constant.

Reflect returns a list of constants with the `getConstants()` method, and each of these elements can be exploited with this `ConstantModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $constants = $package->getConstants();
```

```
}
```

This list (`$constants`) is an iterator, that can be traversed by a simple foreach loop. Each `$constant` element returned is an instance of `ConstantModel`.

Gets an array of constants for the data source.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getConstants() as $constant) {
        printf( 'Processing constant "%s" ...' . PHP_EOL, $constant->getName() );

        if ($constant->isMagic() === true) {
            echo '- Magic constant';
        } else {
            echo '- User constant with value ' . $constant->getValue();
        }
        echo PHP_EOL;
    }
}
```

And lot more. See ConstantModel Reference to learn all features and behaviors.

`ConstantModel` Reference is not yet available.

15.5.10. Includes

The `Bartlett\Reflect\Model\IncludeModel` class reports information about an include.

Reflect returns a list of includes with the `getIncludes()` method, and each of these elements can be exploited with this `IncludeModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $includes = $package->getIncludes();
}
```

This list (`$includes`) is an iterator, that can be traversed by a simple foreach loop. Each `$include` element returned is an instance of `IncludeModel`.

Gets an array of includes for the data source.

```
<?php

use Bartlett\Reflect;

foreach ($reflect->getPackages() as $package) {
```

```
    foreach ($package->getIncludes() as $include) {
        if ($include->isRequire() === true) {
            printf( '- require "%s"', $include->getFilePath() );
        } elseif ($include->isRequireOnce() === true) {
            printf( '- require_once "%s"', $include->getFilePath() );
        } elseif ($include->isInclude() === true) {
            printf( '- include "%s"', $include->getFilePath() );
        } elseif ($include->isIncludeOnce() === true) {
            printf( '- include_once "%s"', $include->getFilePath() );
        }
        echo PHP_EOL;
    }
}
```

And lot more. See IncludeModel Reference to learn all features and behaviors.

`IncludeModel` Reference is not yet available.

15.5.11. Dependencies

The `Bartlett\Reflect\Model\DependencyModel` class reports information about a dependency like class, an interface, a php or extension function.

Reflect returns a list of dependencies with the `getDependencies()` method, and each of these elements can be exploited with this `DependencyModel`.

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

foreach ($reflect->getPackages() as $package) {
    $dependencies = $package->getDependencies();
}
```

This list (`$dependencies`) is an iterator, that can be traversed by a simple foreach loop. Each `$dependency` element returned is an instance of `DependencyModel`.

Gets an array of internal php functions.

```
<?php

use Bartlett\Reflect;

$internalFunctions = array();

foreach ($reflect->getPackages() as $package) {
    foreach ($package->getDependencies() as $dep) {
        $internalFunctions[] = $dep->getName();
    }
}
print_r($internalFunctions);
```

And lot more. See DependencyModel Reference to learn all features and behaviors.

`DependencyModel` Reference is not yet available.

Chapter 16. Plugins

16.1. Events

Reflect uses a Symfony EventDispatcher [http://symfony.com/doc/current/components/event_dispatcher/index.html] Component to allow you to easily extend the features list.

The EventDispatcher component allow Reflect components to communicate with each other by dispatching events and listening to them.

16.1.1. Event Dispatcher

Reflect implement interface `Bartlett\Reflect\Event\DispatcherInterface`. You can add event listeners and event subscribers to this object.

listeners	Callable functions that are registered on an event dispatcher for specific events.
subscribers	Classes that tell an event dispatcher what methods to listen to and what functions on the class to invoke when the event is triggered. Event subscribers subscribe event listeners to an event dispatcher.

16.1.2. Getting an EventDispatcher

You can get the EventDispatcher of `Bartlett\Reflect\Event\DispatcherInterface` by calling the `getEventDispatcher()` method.

Here is an example :

```
<?php

use Bartlett\Reflect;

$reflect = new Reflect;

$ed = $reflect->getEventDispatcher();
```

16.1.3. Adding Event Listeners

After you have the event dispatcher, you can register event listeners that listen to specific events.

Example 16.1. Add a listener that will echo out files when they are parsed

```
<?php

use Bartlett\Reflect;
use Symfony\Component\EventDispatcher\GenericEvent;

$reflect = new Reflect;

$reflect->getEventDispatcher()->addListener(
    'reflect.progress',
    function (GenericEvent $e) {
```

```
        printf(
            'Parsing Data source "%s" in progress ... File "%s"' . PHP_EOL,
            $e['source'],
            $e['file']->getPathname()
        );
    }
);
```

Example 16.2. Add a listener that will exploit each AST [http://en.wikipedia.org/wiki/Abstract_syntax_tree] of file parsed

```
<?php

use Bartlett\Reflect;
use Symfony\Component\EventDispatcher\GenericEvent;

$reflect = new Reflect;

$reflect->getEventDispatcher()->addListener(
    'reflect.success',
    function (GenericEvent $e) {
        $ast = unserialize($e['ast']);
        printf(
            'Parsing Data source "%s", file "%s". AST = %s' . PHP_EOL,
            $e['source'],
            $e['file']->getPathname(),
            print_r($ast, true)
        );

    }
);
```

16.1.4. Event Subscribers

Event subscribers are classes that implement interface `Symfony\Component\EventDispatcher\EventSubscriberInterface`. They are used to register one or more event listeners to methods of the class. Event subscribers tell event dispatcher exactly which events to listen to and what method to invoke on the class.

Reflect plugins follow the event subscribers behaviors. Have a look on AnalyserPlugin :

```
<?php
class AnalyserPlugin implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            'reflect.complete' => 'onReflectComplete',
        );
    }
}
```

This plugin registers event listeners to the `reflect.complete` event of a Reflect parse request.

When the `reflect.complete` event is emitted, the `onReflectComplete` instance method of the plugin is invoked.

16.1.5. Events lifecycle

Event	Action	Informations available
reflect.progess	Before to parse a new file of the data source.	`source` data source identifier or its alias `file` current file parsed in the data source
reflect.success	After parsing the current file (A cached request will not trigger this event)	`source` data source identifier or its alias `file` current file parsed in the data source `ast` the Abstract Syntax Tree [http://en.wikipedia.org/wiki/Abstract_syntax_tree] result of PHP-Parser [https://github.com/nikic/PHP-Parser]
reflect.cache	A previous cached request was found and return the AST.	`source` data source identifier or its alias `file` current file parsed in the data source `ast` the Abstract Syntax Tree [http://en.wikipedia.org/wiki/Abstract_syntax_tree] result of PHP-Parser [https://github.com/nikic/PHP-Parser]
reflect.error	When PHP Parser raise an error	`source` data source identifier or its alias `file` current file parsed in the data source `error` PHP Parser error message
reflect.complete	When a parse request is over.	`source` data source identifier or its alias

16.2. Console Commands

If your plugin should be accessible on the command line, and provides some new commands, you have to register them with the static `getCommands()` method.

Have a look on AnalyserPlugin, that provide two new commands: `analyser:list` and `analyser:run`.

```
<?php
```

```
class AnalyserPlugin implements EventSubscriberInterface
{
    public static function getCommands()
    {
        $commands   = array();
        $commands[] = new AnalyserListCommand;
        $commands[] = new AnalyserRunCommand;

        return $commands;
    }
```

If your plugin must not provide console command, your `getCommands()` static method should return an empty php array.

16.3. Register Plugins

Don’t forget to register a plugin with `addSubscriber()` method, if you want to use it.

Example 16.3. Register the cache plugin

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\Plugin\Cache\CachePlugin;

$reflect = new Reflect;

$reflect->addSubscriber( new CachePlugin($cache) );
```

Learn more about the cache plugin, see Section 8.2.1, “Cache Plugin”

Chapter 17. Cache Plugin

17.1. Register Plugin

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\Plugin\Cache\CachePlugin;

$reflect = new Reflect;
$reflect->addSubscriber( new CachePlugin($cache) );
```

Where `$cache` is an instance of object that must implement interface `Bartlett\Reflect\Plugin\Cache\CacheStorageInterface`.

Use the `Bartlett\Reflect\Plugin\Cache\DefaultCacheStorage` object unless you want to change the cache storage behavior.

17.2. Doctrine Adapter

Use one of the most famous caching solution, provided by the Doctrine project.

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\Plugin\Cache\CachePlugin;
use Bartlett\Reflect\Plugin\Cache\DefaultCacheStorage;
use Bartlett\Reflect\Cache\DoctrineCacheAdapter;

use Doctrine\Common\Cache\FilesystemCache;

$doctrineCache = new DoctrineCacheAdapter($backend);

$cache = new DefaultCacheStorage($doctrineCache);

$reflect = new Reflect;
$reflect->addSubscriber( new CachePlugin($cache) );
```

Where `$backend` is an instance of object that must implement interface `Doctrine\Common\Cache\CacheProvider`.

17.3. File cache

Doctrine File backend to store your Reflect results in the local file system.

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\Plugin\Cache\CachePlugin;
use Bartlett\Reflect\Plugin\Cache\DefaultCacheStorage;
use Bartlett\Reflect\Cache\DoctrineCacheAdapter;
```

```
use Doctrine\Common\Cache\FilesystemCache;

$backend = new FilesystemCache(sys_get_temp_dir() . '/bartlett/cache');

$doctrineCache = new DoctrineCacheAdapter($backend);

$cache = new DefaultCacheStorage($doctrineCache);

$reflect = new Reflect;
$reflect->addSubscriber( new CachePlugin($cache) );
```

In the source code above, we use the standard Doctrine File cache provider, and store results in the default system temporary directory (see php sys_get_temp_dir() [http://www.php.net/manual/en/function.sys-get-temp-dir.php] function).

Chapter 18. Log Plugin

18.1. Register Plugin

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\Plugin\Log\LogPlugin;

// Optional plugin configuration
$opt = array();

$reflect = new Reflect;
$reflect->addSubscriber( new LogPlugin($logger, $opt) );
```

Where `$logger` is an instance of object that must implement interface `Psr\Log\LoggerInterface` (PSR-3 [http://www.php-fig.org/psr/psr-3/]).

And `$opt` is an array to configure what events and its details you would like to have.

Event	Log Level	Message Template (with/ without placeholders)
reflect.progess	Psr3\Log\Level\LogLevel::INFO	Parsing file "{file}" in progress.
reflect.success	Psr3\Log\Level\LogLevel::INFO	AST built.
reflect.cache	Psr3\Log\Level\LogLevel::INFO	AST built by a previous request.
reflect.error	Psr3\Log\Level\LogLevel::ERROR	Parser has detected an error on file "{file}". "{error}".
reflect.complete	Psr3\Log\Level\LogLevel::NOTICE	Parsing data source "{source}" completed.

For example, if you want to deactivate logging on `reflect.success` event, then give the following options :

```
<?php

$opt = array(
    'reflect.success' => false,
);
```

Or, if you don’t want to have contextual data sent to logger :

```
<?php

$opt = array(
    'reflect.success' => array(
        'level'    => LogLevel::INFO,
```

```
        'template' => 'AST built.',
        'context'  => false,
    ),
);
```

18.2. Using your private logger

Use your own logger, that must be compatible PSR-3.

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\Plugin\Log\LogPlugin;

use Psr\Log\AbstractLogger;

class YourLogger extends AbstractLogger
{
    private $channel;

    public function __construct($name = 'YourLoggerChannel')
    {
        $this->channel = $name;
    }

    public function log($level, $message, array $context = array())
    {
        error_log(
            sprintf(
                '%s.%s: %s',
                $this->channel,
                strtoupper($level),
                $this->interpolate($message, $context)
            )
        );
    }

    protected function interpolate($message, array $context = array())
    {
        // build a replacement array with braces around the context keys
        $replace = array();
        foreach ($context as $key => $val) {
            $replace['{' . $key . '}'] = $val;
        }

        // interpolate replacement values into the message and return
        return strtr($message, $replace);
    }
}

// Create the main logger
$logger = new YourLogger('Reflect');

// Optional plugin configuration
$opt = array();

$reflect = new Reflect;
$reflect->addSubscriber( new LogPlugin($logger, $opt) );
```

18.3. Using Monolog

Use one of the most famous logging solution compatible PSR-3.

```
<?php

use Bartlett\Reflect;
use Bartlett\Reflect\Plugin\Log\LogPlugin;

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

// Create some handlers
$stream = new StreamHandler('/var/logs/phpreflect.log');

// Create the main logger
$logger = new Logger('Reflect');
$logger->pushHandler($stream);

// Optional plugin configuration
$opt = array();

$reflect = new Reflect;
$reflect->addSubscriber( new LogPlugin($logger, $opt) );
```

If you want to use Monolog with Reflect on CLI mode, then you should use a wrapper like this.

```
<?php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

class YourLogger extends Logger
{
    public function __construct($name = 'YourLoggerChannel')
    {
        $stream = new StreamHandler('/var/logs/phpreflect.log');
        parent::__construct($name, array($stream));
    }
}
```

Chapter 19. Analysers

Analysers implements the Visitor [http://en.wikipedia.org/wiki/Visitor_pattern] pattern in a simple and effective way to make the render of your results truly customizable.

19.1. Visitor pattern

Each `Analyser` class must implement interface `Bartlett\Reflect\Visitor\VisitorInterface`.

```
<?php

namespace Bartlett\Reflect\Visitor;

use Bartlett\Reflect\Model\Visitable;

interface VisitorInterface
{
    public function visit(Visitable $visitable);
}
```

Each element that need to be explored by your analyser should have a visit method accordingly.

- For packages, we need to implement a **visitPackageModel** method.
- For classes, we need to implement a **visitClassModel** method.
- For properties, we need to implement a **visitPropertyModel** method.
- For methods, we need to implement a **visitMethodModel** method.
- For functions, we need to implement a **visitFunctionModel** method.
- For function or method parameters, we need to implement a **visitParameterModel** method.
- For constants, we need to implement a **visitConstantModel** method.
- For includes, we need to implement a **visitIncludeModel** method.
- For dependencies, we need to implement a **visitDependencyModel** method.

Abstract class `Bartlett\Reflect\Visitor\AbstractVisitor` (source code [https://raw.githubusercontent.com/llaville/php-reflect/master/src/Bartlett/Reflect/Visitor/AbstractVisitor.php]), that implement interface `Bartlett\Reflect\Visitor\VisitorInterface`, holds a basic visitor.

```
<?php

use Bartlett\Reflect\Visitor\AbstractVisitor;

class Analyser extends AbstractVisitor
{
    public function visitPackageModel($package)
    {
```

```
    }

    public function visitUseModel($use)
    {
    }

    public function visitClassModel($class)
    {
    }

    public function visitPropertyModel($property)
    {
    }

    public function visitMethodModel($method)
    {
    }

    public function visitFunctionModel($function)
    {
    }

    public function visitParameterModel($parameter)
    {
    }

    public function visitConstantModel($constant)
    {
    }

    public function visitIncludeModel($include)
    {
    }

    public function visitDependencyModel($dependency)
    {
    }
}
```

An abstract class `Bartlett\Reflect\Analyser\AbstractAnalyser` (source code [https://raw.githubusercontent.com/llaville/php-reflect/master/src/Bartlett/Reflect/Analyser/AbstractAnalyser.php]) that implement all required interfaces may be used to initialize common data in a simple way.

Your analyser became as simple like that:

```
<?php

use Bartlett\Reflect\Analyser\AbstractAnalyser;

class Analyser extends AbstractAnalyser
{
}
```

19.2. Print results

Once you have used visit methods to explore parsing results, you probably want to display it.

To do so, you should implement the `render()` method of `Bartlett\Reflect\Analyser\AnalyserInterface`.

Each analyser is responsible to display results with the `render()` method. You can either used the `Bartlett\Reflect\Printer\Text` helper to produces report line by line, or any other solution.

If you used the printer text helper, your implementation should follow only one rule. Return an array with :

- free data identifier as key.
- data contents are an array with a string format compatible vsprintf [http://www.php.net/manual/en/function.vsprintf.php] as first element, and the value as second element.

Example 19.1. Console lines information (without data)

```
<?php

    $lines['dataSourceAnalysed'] = array(
        '<info>Data Source Analysed</info>%s',
        array(PHP_EOL)
    );

    $lines['methodsScope'] = array(
        '    Scope',
        array()
    );
```

Example 19.2. Console line information with only one value

```
<?php

    $lines['methods'] = array(
        '  Methods                                  %10d',
        array($count['methods'])
    );
```

Example 19.3. Console line information with more than one value

```
<?php

    $lines['nonStaticMethods'] = array(
        '      Non-Static Methods                   %10d (%.2f%%)',
        array(
            $count['nonStaticMethods'],
            $count['methods'] > 0 ? ($count['nonStaticMethods'] / $count['methods']) * 1
        )
    );
```

See source code of Structure Analyser [https://raw.githubusercontent.com/llaville/php-reflect/master/src/Bartlett/Reflect/Analyser/StructureAnalyser.php] as example. :leveloffset: 0