phpspec Documentation

Release 3.x

Konstantin Kudryashov (everzet), Marcello Duarte (_md)

Oct 28, 2017

Contents

Create a `composer.json` file:

```
{
    "require-dev": {
        "phpspec/phpspec": "^4.0"
    },
    "config": {
        "bin-dir": "bin"
    },
    "autoload": {"psr-0": {"": "src"}}
}
```

Follow the instructions on this page to install composer: https://getcomposer.org/download/.

Install **phpspec** with composer:

```
php composer.phar install
```

Start writing specs:

```
bin/phpspec desc Acme/Calculator
```

Learn more from *the documentation*.

CHAPTER 1

Introduction

Spec BDD with phpspec

phpspec is a tool which can help you write clean and working PHP code using behaviour driven development or BDD. BDD is a technique derived from test-first development.

BDD is a technique used at story level and spec level. **phpspec** is a tool for use at the spec level or SpecBDD. The technique is to first use a tool like **phpspec** to describe the behaviour of an object you are about to write. Next you write just enough code to meet that specification and finally you refactor this code.

SpecBDD and TDD

There is no real difference between SpecBDD and TDD. The value of using an xSpec tool instead of a regular xUnit tool for TDD is **the language**. The early adopters of TDD focused on behaviour and design of code. Over time the focus has shifted towards verification and structure. BDD aims to shift the focus back by removing the language of testing. The concepts and features of the tool will keep your focus on the "right" things.

SpecBDD and StoryBDD

StoryBDD tools like Behat help to understand and clarify the domain. They help specify feature narratives, their needs, and what we mean by them. With SpecBDD we are only focused on the how, in other words, the implementation. You are specifying how your classes will achieve those features.

Only using story level BDD will not do enough to help you write the code for the features well. Each feature is likely to need quite a lot of code. If you only confirm that the whole feature works and also only refactor at that point then you are working in large steps. SpecBDD tools guide you in the process by letting you write the code in small steps. You only need to write the spec and then the code for the next small part you want to work on and not the whole feature.

StoryBDD and SpecBDD used together are an effective way to achieve customer-focused software.

CHAPTER 2

Installation

phpspec is a php 5.6+ library that you'll have in your project development environment. Before you begin, ensure that you have PHP 5.6 or 7 installed.

Installation process:

You can install phpspec with all its dependencies through Composer. Follow instructions on the composer website if you don't have it installed yet.

N.b.: You will need to ensure that your Composer `autoload` settings are correct. phpspec will not be able to detect classes, even ones it has created, unless this is working. This is a common issue which causes confusion when installing phpspec.

The `autoload` section of your `composer.json` file may look something like this:

```
"autoload": {
    "psr-0": {
        "": "src/"
    }
}
```

Method #1 (Composer command):

You can use this Composer command to install phpspec:

```
composer require --dev phpspec/phpspec
```

Method #2 (Composer config file):

If you prefer editing your `composer.json` file manually, add phpspec to your `require-dev` section like this:

```
{
    "require-dev": {
        "phpspec/phpspec": "[your preferred version]"
    },
    "config": {
        "bin-dir": "bin"
    },
    "autoload": {
        "psr-0": {
            "": "src/"
        }
    }
}
```

Then install phpspec with the composer install command:

```
$ composer install
```

Result:

phpspec with its dependencies will be installed inside the `vendor` folder. The phpspec executable will be available at `vendor/bin/phpspec`, or wherever you have specified in your `composer.json` file's `bin-dir` setting. See the composer docs for more information

CHAPTER 3

Getting Started

Say you are building a tool that converts Markdown into HTML. Well, that's a large task. But you can work on simple things first and a design will emerge that will reach all the necessary features.

What is the simplest thing you could add? It should convert a string line into a paragraph with HTML markup, i.e. *"Hi, there"* would become *" Hi, there "*.

So you can start by doing this. Well, not the boring bits. Let **phpspec** take care of the boring stuff for you. You just need to tell **phpspec** you will be working on the *Markdown* class.

```
$ bin/phpspec desc Markdown
Specification for Markdown created in spec.
```

You can also specify a fully qualified class name. Don't forget that if you use backslashes you need to pass the class name inside double quotes. Alternatively you could use forward slashes and skip the quotes. **phpspec** will create the folder structure following PSR standards.

Ok. What have you just done? **phpspec** has created the spec for you! You can navigate to the spec folder and see the spec there:

```
<?php

namespace spec;

use Markdown;
use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class MarkdownSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType(Markdown::class);
    }
}
```

So what do you have here? Your spec extends the special `ObjectBehavior` class. This class is special, because it gives you the ability to call all the methods of the class you are describing and match the result of the operations against your expectations.

Examples

The object behavior is made up of examples. Examples are encased in public methods, started with `it_` or `its_`.

phpspec searches for these methods in your specification to run.

Why are underscores used in example names? `just_because_its_much_easier_to_read` than `someLongCamelCasingLikeThat`.

Specifying behaviour

Now we are ready to move on. Let's update that first example to express your next intention:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MarkdownSpec extends ObjectBehavior
{
    function it_converts_plain_text_to_html_paragraphs()
    {
        $this->toHtml("Hi, there")->shouldReturn("<p>Hi, there</p>");
    }
}
```

Here you are telling **phpspec** that your object has a `toHtml` method. You are also telling it that this method should return " Hi, there ". Now what? Run the specs. You may not believe this, but **phpspec** will understand you are describing a class that doesn't exist and offer to create it!

```
$ bin/phpspec run

> spec\Markdown

   it converts plain text to html paragraphs
      Class Markdown does not exist.

         Do you want me to create it for you? [Y/n]
```

phpspec will then place the empty class in the directory. Run your spec again and... OK, you guessed:

```
$ bin/phpspec run

> spec\Markdown

   it converts plain text to html paragraphs
      Method Markdown::toHtml() not found.

         Do you want me to create it for you? [Y/n]
```

What you just did was moving fast through the amber state into the red.

```
<?php

class Markdown
{
    public function toHtml($argument1)
    {
        // TODO: write logic here
    }
}
```

You got rid of the fatal errors and ugly messages that resulted from non-existent classes and methods and went straight into a real failed spec:

```
$ bin/phpspec run

> spec\Markdown

   it converts plain text to html paragraphs
      Expected "<p>Hi, there</p>", but got null.


1 examples (1 failed)
284ms
```

You can change the generated specs and classes using *templates*.

According to the TDD rules you now have full permission to write code. Red means "time to add code"; red is great! Now you can add just enough code to make the spec green, quickly. There will be time to get it right, but first just get it green.

```
<?php

class Markdown
{
    public function toHtml()
    {
        return "<p>Hi, there</p>";
    }
}
```

And voilà:

```
$ bin/phpspec run

> spec\Markdown

   it converts plain text to html paragraphs

1 examples (1 passed)
247ms
```

There are heaps of resources out there already if you would like to read more about the TDD/SpecBDD cycle. Here are just a couple for you to look at:

1. The Rspec Book Development with RSpec, Cucumber, and Friends by David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, Dan North

2. Test Driven Development: By Example Kent Beck

In the example here you specified the value the `toHtml` method should return by using one of **phpspec's** matchers. There are several other matchers available, you can read more about these in the *Matchers Cookbook*

Skipping examples

It may happen that some of your examples will depend on some environment requirements. For example, it might need a php extension or a minimal php version. In that case, you don't want your examples to fail because **phpspec** is unable to run them.

phpspec allows to easily skip an example by throwing a *SkippingException* wherever you feel the need for it.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use PhpSpec\Exception\Example\SkippingException;

class RocketSpec extends ObjectBehavior
{
    function it_flies_around_the_moon()
    {
        if (!function_exists('rocket_launch')) {
            throw new SkippingException(
                'The rocket extension is not installed'
            );
        }
        $this->flyToTheMoon();
    }
}
```

An extension is also available to skip examples regarding a class/interface was not found. It can be found here: https://github.com/akeneo/PhpSpecSkipExampleExtension

CHAPTER 4

Prophet Objects

Stubs

You also need your *Markdown* parser to be able to format text fetched from an external source such as a file. You decide to create an interface so that you can have different implementations for different types of source.

```
<?php

namespace Markdown;

interface Reader
{
    public function getMarkdown();
}
```

You want to describe a method which has an instance of a *Reader* as an argument. It will call `Markdown\Reader::getMarkdown()` to get the markdown to format. You have not yet written any implementations of Reader to pass into the method though. You do not want to get distracted by creating an implementation before you can carry on writing the parser. Instead we can create a fake version of Reader called a stub and tell **phpspec** what `Markdown\Reader::getMarkdown()` should return.

You can create a stub by telling **phpspec** that you want it to be a double of the *MarkdownReader* interface:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MarkdownSpec extends ObjectBehavior
{
    function it_converts_text_from_an_external_source($reader)
    {
        $reader->beADoubleOf('Markdown\Reader');
        $this->toHtmlFromReader($reader)->shouldReturn("<p>Hi, there</p>");
```

```
    }
}
```

At the moment calling `Markdown\Reader::getMarkdown()` will return null. We can tell **phpspec** what we want it to return though.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MarkdownSpec extends ObjectBehavior
{
    function it_converts_text_from_an_external_source($reader)
    {
        $reader->beADoubleOf('Markdown\Reader');
        $reader->getMarkdown()->willReturn("Hi, there");

        $this->toHtmlFromReader($reader)->shouldReturn("<p>Hi, there</p>");
    }
}
```

Now you can write the code that will get this example to pass. As well as refactoring your implementation you should see if you can refactor your specs once they are passing. In this case we can tidy it up a bit as **phpspec** lets you create the stub in an easier way. If you use a typehint, **phpspec** determine the required type of the collaborator:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Reader;

class MarkdownSpec extends ObjectBehavior
{
    function it_converts_text_from_an_external_source(Reader $reader)
    {
        $reader->getMarkdown()->willReturn("Hi, there");

        $this->toHtmlFromReader($reader)->shouldReturn("<p>Hi, there</p>");
    }
}
```

phpspec 2.* supports the use of *@param* annotations instead of parametric typehints for this purpose. However, this functionality is removed in **phpspec** 3.0.

Mocks

You also need to be able to get your parser to output to somewhere instead of just returning the formatted text. Again you create an interface:

```
<?php

namespace Markdown;
```

```
interface Writer
{
    public function writeText($text);
}
```

You again pass it to the method but this time the `Markdown\Writer::writeText($text)` method does not return something to your parser class. The new method you are going to create on the parser will not return anything either. Instead it is going to give the formatted text to the *MarkdownWriter* so you want to be able to give an example of what that formatted text should be. You can do this using a mock, the mock gets created in the same way as the stub. This time you tell it to expect `Markdown\Writer::writeText($text)` to get called with a particular value:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function it_outputs_converted_text(Writer $writer)
    {
        $writer->writeText("<p>Hi, there</p>")->shouldBeCalled();

        $this->outputHtml("Hi, there", $writer);
    }
}
```

Now if the method is not called with that value then the example will fail.

Spies

Instead of using a mock you could use a spy. The difference is that you check what happened after the object's behaviour has happened:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function it_outputs_converted_text(Writer $writer)
    {
        $this->outputHtml("Hi, there", $writer);

        $writer->writeText("<p>Hi, there</p>")->shouldHaveBeenCalled();
    }
}
```

The difference is one of style. You may prefer to use mocks and say what should happen beforehand. You may prefer to use spies and say what should have happened afterwards.

CHAPTER 5

Let and Let Go

If you need to pass the object into the constructor instead of a method then you can do it like this:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function it_outputs_converted_text(Writer $writer)
    {
        $this->beConstructedWith($writer);
        $writer->writeText("<p>Hi, there</p>")->shouldBeCalled();

        $this->outputHtml("Hi, there");
    }
}
```

If you have many examples then writing this in each example will get tiresome. You can instead move this to a *let* method. The *let* method gets run before each example so each time the parser gets constructed with a fresh mock object.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function let(Writer $writer)
    {
```

```
        $this->beConstructedWith($writer);
    }
}
```

There is also a *letGo* method which runs after each example if you need to clean up after the examples.

It looks like you will now have difficulty getting hold of the instance of the mock object in the examples. This is easier to deal with than it looks though. Providing you use the same variable name for both, **phpspec** will inject the same instance into the *let* method and the example. The following will work:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function let(Writer $writer)
    {
        $this->beConstructedWith($writer);
    }

    function it_outputs_converted_text($writer)
    {
        $writer->writeText("<p>Hi, there</p>")->shouldBeCalled();

        $this->outputHtml("Hi, there");
    }
}
```

CHAPTER 6

Upgrading to PhpSpec 4

Here is a guide to upgrading a test suite or an extension, based on BC-breaking changes made in phpspec 4.

Upgrading for Users

If you are using 3rd party **phpspec** extensions, you may have to increase the version numbers for those as well.

As PHP 5 is no longer supported language versions, you will need to upgrade to PHP 7 to use **phpspec** 4.

Most methods in PhpSpec now have static type hints and return types, this will affect you when you are overriding behaviour from a parent class or implementing an interface.

If you are providing inline matchers in your specs you will need to provide the array type hint:

```
function getMatchers()
{
    // return some matchers
}
```

Change to:

```
function getMatchers() : array
{
    // return some matchers
}
```

If you are providing custom matchers, you will need to conform to the type hint changes in the Matcher interface.

Upgrading for Extension Authors

Many PhpSpec interfaces and internal classes have had scalar typehints and return typehinting added. You will need to update your implementations of these interfaces to the new method signature.

CHAPTER 7

Upgrading to phpspec 3

Here is a guide to upgrading a test suite or an extension, based on BC-breaking changes made in phpspec 3.

Upgrading for Users

If you are using 3rd party **phpspec** extensions, you may have to increase the version numbers for those as well.

As PHP 5.5 and below are no longer supported language versions, you will need to upgrade to PHP 5.6 or 7.0+ to use **phpspec** 3.

Where you have used *@param* annotations for spec examples, to indicate the required type for a collaborator, you will need to remove these and use explicit typehinting in the method signature instead. For example:

```
/**
 * @param \stdClass $collaborator
 */
function it_does_something_with_a_stdclass($collaborator)
```

Change to:

```
function it_does_something_with_a_stdclass(\stdClass $collaborator)
```

Extension configured in your `phpspec.yml` needs to be changed from:

```
some_extension_config: foo

extensions:
    - SomeExtension
    - SomeOtherExtension
```

To:

```
extensions:
    SomeExtension:
```

```
        some_config: foo

    SomeOtherExtension: ~
```

Upgrading for Extension Authors

Several interfaces have been renamed in **phpspec** 3.0. Here is a quick guide to changes you will need to make in your code.

- `PhpSpec\Console\IO` is now `PhpSpec\Console\ConsoleIO`
- `PhpSpec\IO\IOInterface` is now `PhpSpec\IO\IO`
- `PhpSpec\Locator\ResourceInterface` is now `PhpSpec\Locator\Resource`
- `PhpSpec\Locator\ResourceLocatorInterface` is now `PhpSpec\Locator\ResourceLocator`
- `PhpSpec\Formatter\Presenter\PresenterInterface` is now `PhpSpec\Formatter\Presenter\Presenter`
- `PhpSpec\CodeGenerator\Generator\GeneratorInterface` is now `PhpSpec\CodeGenerator\Generator\Generator`
- `PhpSpec\Extension\ExtensionInterface` is now `PhpSpec\Extension`
- `Phpspec\CodeAnalysis\AccessInspectorInterface` is now `Phpspec\CodeAnalysis\AccessInspector`
- `Phpspec\Event\EventInterface` is now `Phpspec\Event\PhpSpecEvent`
- `PhpSpec\Formatter\Presenter\Differ\EngineInterface` is now `PhpSpec\Formatter\Presenter\Differ\DifferEngine`
- `PhpSpec\Matcher\MatcherInterface` is now `PhpSpec\Matcher\Matcher`
- `PhpSpec\Matcher\MatchersProviderInterface` is now `PhpSpec\Matcher\MatchersProvider`
- `PhpSpec\SpecificationInterface` is now `PhpSpec\Specification`
- `PhpSpec\Runner\Maintainer\MaintainerInterface` is now `PhpSpec\Runner\Maintainer\Maintainer`

Some methods have a different signature:

- `PhpSpec\CodeGenerator\Generator\PromptingGenerator#__construct`'s third and fourth arguments are now mandatory
- `PhpSpec\Matcher\ThrowMatcher#__construct`'s third argument is now mandatory
- `PhpSpec\Extension#load` has now an additional mandatory `array $params` argument.

A few methods have been renamed in **phpspec** 3.0:

- `PhpSpec\ServiceContainer#set` is now `PhpSpec\ServiceContainer#define`
- `PhpSpec\ServiceContainer#setShared` is now `PhpSpec\ServiceContainer#define`

Other things to bear in mind:

- `PhpSpec\ServiceContainer` is now an interface (available implementation: `PhpSpec\ServiceContainer\IndexedServiceContainer`)
- `PhpSpec\ServiceContainer\ServiceContainer#getByPrefix` has been replaced by `PhpSpec\ServiceContainer\ServiceContainer#getByTag`. Tags can be set via `PhpSpec\ServiceContainer\ServiceContainer#define`'s third argument

CHAPTER 8

Configuration

Some things in phpspec can be configured in a `phpspec.yml`, `.phpspec.yml`, or `phpspec.yml.dist` file in the root of your project (the directory where you run the `phpspec` command).

You can use a different config file name and path with the `--config` option:

```
$ bin/phpspec run --config path/to/different-phpspec.yml
```

You can also specify default values for config variables across all repositories by creating the file `.phpspec.yml` in your home folder (Unix systems). phpspec will use your personal preference for all settings that are not defined in the project's configuration.

PSR-4

phpspec can try to autodetect your naming scheme by querying Composer for autoload rules you can define in the Composer manifest. If unsuccessful, it assumes a PSR-0 mapping of namespaces to the src and spec directories by default. So for example running:

```
$ bin/phpspec describe Acme/Text/Markdown
```

Will create a spec in the `spec/Acme/Text/MarkdownSpec.php` file and the class will be created in `src/Acme/Text/Markdown.php`

To use PSR-4 you configure the `namespace` and `psr4_prefix` options in a suite to the part that should be omitted from the directory structure:

```
suites:
    acme_suite:
        namespace: Acme\Text
        psr4_prefix: Acme\Text
```

With this config running:

```
$ bin/phpspec describe Acme/Text/Markdown
```

will now put the spec in `spec/MarkdownSpec.php` and the class will be created in `src/Markdown.php`.

Alternatively, you can choose to use Composer to provide the necessary configuration:

```
composer_suite_detection: true # translates to:
                               # - root_directory: '.'
                               # - spec_prefix: spec
```

Spec and source locations

The default locations used by **phpspec** for the spec files and source files are *spec* and *src* respectively. You may find that this does not always suit your needs. You can specify an alternative location in the configuration file. You cannot do this at the command line as it does not make sense for a spec or source files path to change at runtime.

You can specify alternative values depending on the namespace of the class you are describing. In phpspec, you can group specification files by a certain namespace in a *suite*. For each suite, you have several configuration settings:

- `namespace` - The namespace of the classes. Used for generating spec files, locating them and generating code;
- `spec_prefix` [**default**: `spec`] - The namespace prefix for specifications. The complete namespace for specifications is `%spec_prefix%\%namespace%`;
- `src_path` [**default**: `src`] - The path to store the generated classes. By default paths are relative to the location where **phpspec** was invoked. **phpspec** creates the directories if they do not exist. This does not include the namespace directories;
- `spec_path` [**default**: `.`] - The path of the specifications. This does not include the spec prefix or namespace.
- `psr4_prefix` [**default**: `null`] - A PSR-4 prefix to use.

Some examples:

```
suites:
    acme_suite:
        namespace: Acme\Text
        spec_prefix: acme_spec

    # shortcut for
    # my_suite:
    #     namespace: The\Namespace
    my_suite: The\Namespace
```

Tip: You may use `%paths.config%` in `src_path` and `spec_path` making paths relative to the location of the config file.

Some examples:

```
suites:
    acme_suite:
        namespace: Acme\Text
        spec_prefix: acme_spec
        src_path: %paths.config%/src
        spec_path: %paths.config%
```

phpspec will use suite settings based on the namespaces. If you have suites with different spec directories then `phpspec run` will run the specs from each of the directories using the relevant suite settings.

When you use `phpspec desc` **phpspec** creates the spec using the matching configuration. E.g. `phpspec desc Acme/Text/MyClass` will use the the namespace `acme_spec\Acme\Text\MyClass`.

If the namespace does not match one of the namespaces in the suites config then **phpspec** uses the default settings. If you want to change the defaults then you can add a suite without specifying the namespace.

```
suites:
    #...
    default:
        spec_prefix: acme_spec
        spec_path: acmes-specs
        src_path: acme-src
```

You can just set this suite if you wanted to override the default settings for all namespaces. Since **phpspec** matches on namespaces you cannot specify more than one set of configuration values for a null namespace. If you do add more than one suite with a null namespace then **phpspec** will use the last one defined.

Note that the default spec directory is `.`, specs are created in the *spec* directory because it is the first part of the spec namespace. This means that changing the *spec_path* will result in additional directories before *spec* not instead of it. For example, with the config:

```
suites:
    acme_suite:
        namespace: Acme\Text
        spec_prefix: acme_spec
```

running:

```
$ bin/phpspec describe Acme/Text/Markdown
```

will create the spec in the file `acme_spec/spec/Acme/Text/MarkdownSpec.php`

Formatter

You can also set another default formatter instead of `progress`. The `--format` option of the command can override this setting. To set the formatter, use `formatter.name`:

```
formatter.name: pretty
```

The formatters available by default are:

- progress (default)
- html/h
- pretty
- junit
- dot
- tap

More formatters can be added by *extensions*.

Options

You can turn off code generation in your config file by setting `code_generation`:

```
code_generation: false
```

You can also set your tests to stop on failure by setting `stop_on_failure`:

```
stop_on_failure: true
```

Extensions

To register phpspec extensions, use the `extensions` option. This is an array of extension classes:

```
extensions:
    - PhpSpec\Symfony2Extension\Extension
```

Custom matchers

You may want to make custom matchers available in all specs. Custom matchers can be registered by extension, but there is a simplier way: use the `matchers` setting and provide an array of matcher classes. Each of them must implement `PhpSpec\Matcher\Matcher` interface:

```
matchers:
    - Acme\Matchers\ValidJsonMatcher
    - Acme\Matchers\PositiveIntegerMatcher
```

Bootstrapping

There are times when you would be required to load classes and execute additional statements that the Composer-generated autoloader may not provide, which is likely for a legacy project that wants to introduce phpspec for designing new classes that may rely on some legacy collaborators.

To load a custom bootstrap when running phpspec, use the `bootstrap` setting:

```
bootstrap: path/to/different-bootstrap.php
```

This setting should be in the root of the config file (i.e. not nested under `suites` or anything else).

CHAPTER 9

Running phpspec

The phpspec console command uses Symfony's console component. This means that it inherits the default Symfony console command and options.

phpspec has an additional global option to let you specify a config file other than *phpspec.yml* or *phpspec.yml.dist*:

```
$ bin/phpspec run --config path/to/different-phpspec.yml
```

or:

```
$ bin/phpspec run -c path/to/different-phpspec.yml
```

Read more about this in the *Configuration Cookbook*

Also of note is that using the `--no-interaction` option means that no code generation will be done.

phpspec has the global option to let you specify a custom bootstrap or autoloading script.

```
$ bin/phpspec run --bootstrap=path/to/different-bootstrap.php
```

or:

```
$ bin/phpspec run -b path/to/different-bootstrap.php
```

Describe Command

The `describe` command creates a specification for a class:

```
$ bin/phpspec describe ClassName
```

Will generate a specification ClassNameSpec in the spec directory.

```
$ bin/phpspec describe Namespace/ClassName
```

Will generate a namespaced specification NamespaceClassNameSpec. Note that / is used as the separator. To use \ it must be quoted:

```
$ bin/phpspec describe "Namespace\ClassName"
```

The `describe` command has no additional options. It will create a spec class in the *spec* directory. To configure a different path to the specs you can use *suites* in the configuration file.

Run Command

The `run` command runs the specs:

```
$ bin/phpspec run
```

Will run all the specs in the *spec* directory.

```
$ bin/phpspec run spec/ClassNameSpec.php
```

Will run only the ClassNameSpec.

```
$ bin/phpspec run spec/ClassNameSpec.php:56
```

Will run only specification defined in the ClassNameSpec on line 56.

You can run just the specs in a directory with:

```
$ bin/phpspec run spec/Markdown
```

Which will run any specs found in *spec/Markdown* and its subdirectories. Note that it is the spec location and not namespaces that are used to decide which specs to run. Any spec which has a namespace which does not match its file path will be ignored.

By default, you will be asked whether missing methods and classes should be generated. You can suppress these prompts and automatically choose not to generate code with:

```
$ bin/phpspec run --no-code-generation
```

You can choose to stop on failure and avoid running the remaining specs with:

```
$ bin/phpspec run --stop-on-failure
```

TDD work cycle can be described using three steps: Fail, Pass, Refactor. If you create a failing spec for a new method, the next step will be to make it pass. The easiest way to achieve it, is to simply hard code the method, so it returns the expected value.

phpspec can do that for you.

You can opt to automatically fake return values with:

```
$ bin/phpspec run --fake
```

You can choose the output format with the `--format` option e.g.:

```
$ bin/phpspec run --format=dot
```

The formatters available by default are:

- progress (default)
- html
- pretty
- junit
- dot

More formatters can be added by *extensions*.

CHAPTER 10

Object Construction

In **phpspec** specs the object you are describing is not a separate variable but is *$this*. So instead of writing something like:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MarkdownSpec extends ObjectBehavior
{
    function it_converts_plain_text_to_html_paragraphs()
    {
        $markdown = new Markdown();
        $markdown->toHtml("Hi, there")->shouldReturn("<p>Hi, there</p>");
    }
}
```

as you might with other tools, you write:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MarkdownSpec extends ObjectBehavior
{
    function it_converts_plain_text_to_html_paragraphs()
    {
        $this->toHtml("Hi, there")->shouldReturn("<p>Hi, there</p>");
    }
}
```

On consequence this means that you do not construct the object you are describing in the examples. Instead **phpspec** handles the creation of the object you are describing when you run the specs.

The default way **phpspec** does this is the same as `new Markdown()`. If it does not need any values or dependencies to be passed to it then this is fine but for many objects this will not be good enough. You can tell **phpspec** how you want it to create the object though.

Using the Constructor

You can tell **phpspec** to pass values to the constructor when it constructs the object:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function it_outputs_converted_text(Writer $writer)
    {
        $this->beConstructedWith($writer);
        $writer->writeText("<p>Hi, there</p>")->shouldBeCalled();

        $this->outputHtml("Hi, there");
    }
}
```

Using a Factory Method

You may not want to use the constructor but use static factory methods to create the class. This allows you to create it in different ways for different use cases since you can only have a single constructor in PHP.

```
<?php

use Markdown\Writer;

class Markdown
{
    public static function createForWriting(Writer $writer)
    {
        $markdown = new Self();
        $markdown->writer = $writer;

        return $markdown;
    }
}
```

You can tell **phpspec** this is how you want to construct the object as follows:

```
<?php

namespace spec;
```

```
use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function it_outputs_converted_text(Writer $writer)
    {
        $this->beConstructedThrough('createForWriting', [$writer]);
        $writer->writeText("<p>Hi, there</p>")->shouldBeCalled();

        $this->outputHtml("Hi, there");
    }
}
```

Where the first argument is the method name and the second an array of the values to pass to that method.

To be more descriptive, shorter syntaxes are available. All of the following are equivalent:

```
$this->beConstructedNamed('Bob');
$this->beConstructedThroughNamed('Bob');
$this->beConstructedThrough('Named', array('Bob'));
```

Overriding

To avoid repetition you can tell **phpspec** how to construct the object in *let*. However, you may have a single example that needs constructing in a different way. You can do this by calling `beConstructedWith` again in the example. The last time you call `beConstructedWith` will determine how **phpspec** constructs the object:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Markdown\Writer;

class MarkdownSpec extends ObjectBehavior
{
    function let(Writer $writer)
    {
        $this->beConstructedWith($writer, true);
    }

    function it_outputs_converted_text(Writer $writer)
    {
        // constructed with second argument set to true
        // ...
    }

    function it_does_something_if_argument_is_false(Writer $writer)
    {
        $this->beConstructedWith($writer, false);
        // constructed with second argument set to false
        // ...
    }
}
```

CHAPTER 11

Matchers

You use matchers in **phpspec** to describe how an object should behave. They are like assertions in xUnit but with a focus on specifying behaviour instead of verifying output. You use the matchers prefixed by `should` or `shouldNot` as appropriate.

phpspec has 14 built-in matchers, described in more detail here. Many of these matchers have aliases which you can use to make your specifications easy to read.

Custom matchers classes can be registered in configuration.

Identity Matcher

If you want to specify that a method returns a specific value, you can use the Identity matcher. It compares the result using the identity operator: `===`.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_be_a_great_movie()
    {
        $this->getRating()->shouldBe(5);
        $this->getTitle()->shouldBeEqualTo("Star Wars");
        $this->getReleaseDate()->shouldReturn(233366400);
        $this->getDescription()->shouldEqual("Inexplicably popular children's film");
    }
}
```

All four of these ways of using the Identity matcher are equivalent. There is no difference in how they work, this lets you choose the one which makes your specification easier to read.

Comparison Matcher

The Comparison matcher is like the Identity matcher. The difference is that is uses the comparison operator ==. So it is not as strict and follows the PHP rules for loose type comparison.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_be_a_great_movie()
    {
        $this->getRating()->shouldBeLike('5');
    }
}
```

Using `shouldBeLike` it does not matter whether `StarWars::getRating()` returns an integer or a string. The spec will pass for 5 and "5".

Approximately Matcher

If you want to specify that a method returns a value that approximates to a certain precision the given value, you can use the Approximately matcher.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_return_a_near_value()
    {
        $this->getRating()->shouldBeApproximately(1.444447777, 1.0e-9);
        $this->getRating()->shouldBeEqualToApproximately(1.444447777, 1.0e-9);
        $this->getRating()->shouldEqualApproximately(1.444447777, 1.0e-9);
        $this->getRating()->shouldReturnApproximately(1.444447777, 1.0e-9);
    }
}
```

The first argument is the value we expect, the second is the delta.

All four of these ways of using the Approximately matcher are equivalent. There is no difference in how they work, this lets you choose the one which makes your specification easier to read.

Throw Matcher

You can describe an object throwing an exception using the Throw matcher. You use the Throw matcher by calling it straight from `$this`, making the example easier to read.

```php
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_not_allow_negative_ratings()
    {
        $this->shouldThrow('\InvalidArgumentException')->duringSetRating(-3);
    }
}
```

You can also write this as:

```php
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_not_allow_negative_ratings()
    {
        $this->shouldThrow('\InvalidArgumentException')->during('setRating', array(-
↪3));
    }
}
```

The first argument of `during` is the method name and the second one is an array of values passed to the method.

You may want to specify the message of the exception. You can do this by passing an exception object to shouldThrow:

```php
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_not_allow_negative_ratings()
    {
        $this->shouldThrow(new \InvalidArgumentException("Invalid rating"))->during(
↪'setRating', array(-3));
    }
}
```

If you want to use the Throw matcher to check for exceptions thrown during object instantiation you can use the `duringInstantiation` method.

```php
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
```

```
class MovieSpec extends ObjectBehavior
{
    function it_should_not_allow_negative_ratings()
    {
        $this->beConstructedWith(-3);
        $this->shouldThrow('\InvalidArgumentException')->duringInstantiation();
    }
}
```

You can also use the Throw matcher with named constructors.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_not_allow_negative_ratings()
    {
        $this->beConstructedThrough('rated', array(-3));
        $this->shouldThrow('\InvalidArgumentException')->duringInstantiation();
    }
}
```

Trigger Matcher

Let's say you have the following class and a method which is deprecated

```
<?php

class Movie
{
    function setStars($value)
    {
        trigger_error('The method setStars is deprecated. Use setRating instead', E_
↪USER_DEPRECATED);

        $this->rating = $value * 4;
    }
}
```

You can describe an object triggering an error using the Trigger matcher. You use the Trigger matcher by calling it straight from `$this`, making the example easier to read.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
```

```
    function set_stars_should_be_deprecated()
    {
        $this->shouldTrigger(E_USER_DEPRECATED)->duringSetStars(4);
    }
}
```

You may want to specify the message of the error. You can do this by adding a string parameter to the *shouldTrigger* method :

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function set_stars_should_be_deprecated()
    {
        $this->shouldTrigger(E_USER_DEPRECATED, 'The method setStars is deprecated.␣
↪Use setRating instead')->duringSetRating(4);
    }
}
```

Note: As with the Throw matcher, you can also use the *during* syntax described in the Throw section, or use the instantiation mechanisms (such as duringInstantiation, ... etc)

Type Matcher

You can specify the type of the object you are describing with the Type matcher. You can also use this matcher to check that a class implements an interface or that it extends a class.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_be_a_movie()
    {
        $this->shouldHaveType('Movie');
        $this->shouldReturnAnInstanceOf('Movie');
        $this->shouldBeAnInstanceOf('Movie');
        $this->shouldImplement('Movie');
    }
}
```

All four matcher methods are equivalent and will serve to describe if the object is a `Movie` or not.

ObjectState Matcher

The ObjectState matcher lets you check the state of an object by calling methods on it. These methods should start with `is*` or `has*` and return a boolean.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_be_available_on_cinemas()
    {
        // calls isAvailableOnCinemas()
        $this->shouldBeAvailableOnCinemas();
    }

    function it_should_have_soundtrack()
    {
        // calls hasSoundtrack()
        $this->shouldHaveSoundtrack();
    }
}
```

The spec will pass if the object has `isAvailableOnCinemas` and `hasSoundtrack` methods which both return true:

```
<?php

class Movie
{
    public function isAvailableOnCinemas()
    {
        return true;
    }

    public function hasSoundtrack()
    {
        return true;
    }
}
```

Count Matcher

You can check the number of items in the return value using the Count matcher. The returned value could be an array or an object that implements the `\Countable` or `\Traversable` interface.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
```

```
class MovieSpec extends ObjectBehavior
{
    function it_should_have_one_director()
    {
        $this->getDirectors()->shouldHaveCount(1);
    }
}
```

Scalar Matcher

To specify that the value returned by a method should be a specific primitive type you can use the Scalar matcher. It's like using one of the `is_*` functions, e.g, `is_bool`, `is_integer`, `is_float`, etc.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_a_string_as_title()
    {
        $this->getTitle()->shouldBeString();
    }

    function it_should_have_an_array_as_cast()
    {
        $this->getCast()->shouldBeArray();
    }
}
```

IterableContain Matcher

You can specify that a method should return an array or an implementor of `\Traversable` that contains a given value with the IterableContain matcher. **phpspec** matches the value by identity (`===`).

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_contain_jane_smith_in_the_cast()
    {
        $this->getCast()->shouldContain('Jane Smith');
    }
}
```

IterableKeyWithValue Matcher

This matcher lets you assert a specific value for a specific key on a method that returns an array or an implementor of `\ArrayAccess` or `\Traversable`. **phpspec** matches both the key and value by identity (===).

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_jane_smith_in_the_cast_with_a_lead_role()
    {
        $this->getCast()->shouldHaveKeyWithValue('leadRole', 'John Smith');
    }
}
```

IterableKey Matcher

You can specify that a method should return an array or an object implementing `\ArrayAccess` or `\Traversable` with a specific key using the IterableKey matcher. **phpspec** matches the key by identity (===).

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_a_release_date_for_france()
    {
        $this->getReleaseDates()->shouldHaveKey('France');
    }
}
```

IterateAs Matcher

This matcher lets you specify that a method should return an array or an object implementing `\Traversable` that iterates just as the argument you passed to it. **phpspec** matches both the key and the value by identity (===).

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_contain_jane_smith_and_john_smith_in_the_cast()
```

```
    {
        $this->getCast()->shouldIterateAs(new \ArrayIterator(['Jane Smith', 'John␣
↪Smith']));
        $this->getCast()->shouldYield(new \ArrayIterator(['Jane Smith', 'John Smith
↪']));
    }
}
```

Both of these ways of using the IterateAs matcher are equivalent. There is no difference in how they work, this lets you choose the one which makes your specification easier to read.

IterateLike Matcher

This matcher lets you specify that a method should return an array or an object implementing `\Traversable` that iterates equal to the arguments you passed to it. **phpspec** matches both the key and the element by value (==).

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_contain_jane_smith_and_john_smith_in_the_cast()
    {
        $this->getCast()->shouldIterateLike(new \ArrayIterator(['Jane Smith', 'John␣
↪Smith']));
        $this->getCast()->shouldYieldLike(new \ArrayIterator(['Jane Smith', 'John␣
↪Smith']));
    }
}
```

Both of these ways of using the IterateAs matcher are equivalent. There is no difference in how they work, this lets you choose the one which makes your specification easier to read.

StartIteratingAs Matcher

This matcher lets you specify that a method should return an array or an object implementing `\Traversable` that starts iterating just as the argument you passed to it. **phpspec** matches both the key and the value by identity (===).

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_contain_at_least_jane_smith_in_the_cast()
    {
        $this->getCast()->shouldStartIteratingAs(new \ArrayIterator(['Jane Smith']));
        $this->getCast()->shouldStartYielding(new \ArrayIterator(['Jane Smith']));
```

```
    }
}
```

Both of these ways of using the StartIteratingAs matcher are equivalent. There is no difference in how they work, this lets you choose the one which makes your specification easier to read.

StringContain Matcher

The StringContain matcher lets you specify that a method should return a string containing a given substring. This matcher is case sensitive.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_a_title_that_contains_wizard()
    {
        $this->getTitle()->shouldContain('Wizard');
    }
}
```

StringStart Matcher

The StringStart matcher lets you specify that a method should return a string starting with a given substring.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_a_title_that_starts_with_the_wizard()
    {
        $this->getTitle()->shouldStartWith('The Wizard');
    }
}
```

StringEnd Matcher

The StringEnd matcher lets you specify that a method should return a string ending with a given substring.

```
<?php

namespace spec;
```

```
use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_a_title_that_ends_with_of_oz()
    {
        $this->getTitle()->shouldEndWith('of Oz');
    }
}
```

StringRegex Matcher

The StringRegex matcher lets you specify that a method should return a string matching a given regular expression.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_a_title_that_contains_wizard()
    {
        $this->getTitle()->shouldMatch('/wizard/i');
    }
}
```

Inline Matcher

You can create custom matchers by providing them in `getMatchers` method.

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_some_specific_options_by_default()
    {
        $this->getOptions()->shouldHaveKey('username');
        $this->getOptions()->shouldHaveValue('diegoholiveira');
    }

    public function getMatchers(): array
    {
        return [
            'haveKey' => function ($subject, $key) {
                return array_key_exists($key, $subject);
            },
```

```
            'haveValue' => function ($subject, $value) {
                return in_array($value, $subject);
            },
        ];
    }
}
```

In order to print a more verbose error message your inline matcher should throw *FailureException*:

```
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use PhpSpec\Exception\Example\FailureException;

class MovieSpec extends ObjectBehavior
{
    function it_should_have_some_specific_options_by_default()
    {
        $this->getOptions()->shouldHaveKey('username');
        $this->getOptions()->shouldHaveValue('diegoholiveira');
    }

    public function getMatchers(): array
    {
        return [
            'haveKey' => function ($subject, $key) {
                if (!array_key_exists($key, $subject)) {
                    throw new FailureException(sprintf(
                        'Message with subject "%s" and key "%s".',
                        $subject, $key
                    ));
                }
                return true;
            }
        ];
    }
}
```

CHAPTER 12

Templates

phpspec can generate code snippets that will save you time when specifying classes. The default templates will be suitable for many use cases.

However in some cases, it'll be useful to customize those templates by providing ones that suit your project requirements. For example, you may need to add licence information in a docblock to every class file. Instead of doing this manually you can modify the template so it is already in the generated file.

Overriding templates

phpspec uses three templates:

- *specification* - used when a spec is generated using the *describe* command
- *class* - used to generate a class that is specified but which does not exist
- *method* - used to add a method that is specified to a class

You can override these on a per project basis by creating a template file in *.phpspec* in the root directory of the project. For example, to add licence information to the docblock for a class, you can create a file `{project_directory}/.phpspec/class.tpl`. You can copy the contents of the default template found in **phpspec** at `src/PhpSpec/CodeGenerator/Generator/templates/class.template` and add the docblock to it:

```
<?php

/*
 * This file is part of Acme.
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */%namespace_block%

class %name%
{
}
```

So now, for example, you want to describe a class `Acme\Model\Foo` which does not exist. You can run the spec `spec/Acme/Model/FooSpec.php` and let **phpspec** generate the missing class. **phpspec** will use your overridden template and the generated file will look like:

```
<?php

/*
 * This file is part of Acme.
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Acme\Model;

class Foo
{
}
```

You can also override the templates for all your projects by creating a template in *.phpspec* in your home directory.

phpspec uses the first template it finds by looking in this order:

1. `{project_directory}/.phpspec/{template_name}.tpl`
2. `{home_directory}/.phpspec/{template_name}.tpl`
3. The default template

Parameters

As well as static text there are some parameters available like the `%namespace_block%` in the example above. The parameters available depend on which type of template you are overriding:

specification

- `%filepath%` the file path of the class
- `%name%` the specification name
- `%namespace%` the specification namespace
- `%subject%` the fully-qualified name of the class being specified
- `%subject_class%` the name of the class being specified

class

- `%filepath%` the file path of the class
- `%name%` the class name
- `%namespace%` the class namespace
- `%namespace_block%` the formatted class namespace

method

- `%name%` the method name
- `%arguments%` the method arguments

CHAPTER 13

Extensions

Extensions can add functionality to **phpspec**, such as, integration with a particular framework. See below for some example extensions.

Installation

Individual extensions will have their own documentation that you can follow. Usually you can install an extension by adding it to your `composer.json` file and updating your vendors.

Configuration

You will need to tell **phpspec** that you want to use the extension. You can do this by adding it to the config file:

```
extensions:
    MageTest\PhpSpec\MagentoExtension\Extension: ~
```

You can pass options to the extension as well:

```
extensions:
    MageTest\PhpSpec\MagentoExtension\Extension:
        mage_locator:
            src_path: public/app/code
            spec_path: spec/public/app/code
```

See the *Configuration Cookbook* for more about config files.

Example extensions

Framework Integration

- Symfony2
- Magento
- Laravel

Code generation

- Typehinted Methods
- Example Generation
- SpecGen

Additional Formatters

- Nyan Formatters

Metrics

- Code coverage

Matchers

- Coduo matcher extension
- Array Contains matcher extension
- Collection of custom matchers

Miscellaneous

- Prepare
- Data provider
- Behat Integration
- Example skipping through annotation

CHAPTER 14

Working with Wrapped Objects

phpspec wraps some of the objects used in specs. For example `$this` is the object you are describing wrapped in a phpspec object. This is how you can call methods on `$this` and then call matchers on the returned values.

Most of the time this is not something you need to worry about but sometimes it can be an issue.

If you ever need to get the actual object then you can by calling `$this->getWrappedObject()`.

If you try to specify a method on your object that starts with "should", for example:

```
function it_should_handle_something($somethingToHandle)
{
    $this->shouldHandle($somethingToHandle);
}
```

Then this will not work as expected because **phpspec** will intercept the call thinking it is a matcher. You can avoid this by using `callOnWrappedObject`:

```
function it_should_handle_something($somethingToHandle)
{
    $this->callOnWrappedObject('shouldHandle', array($somethingToHandle));
}
```