Nek

Zeste de savoir

Les tests automatisés avec phpspec

10 juillet 2020

Table des matières

Vous cherchez un moyen simple et efficace de tester votre code php ? Ce cours est ce qu'il vous faut. Nous allons expliquer étape par étape comment tester du code en utilisant **phpspec** .

Notez que phpspec est très orienté « SDD » et ne fait en aucun cas de test d'intégration de votre code dans votre application complète.

SDD ? Ça veut dire Specification Driven Development. Pour faire simple, c'est écrire une spécification avant d'écrire le code. Avec phpspec, c'est simple ! Nous verrons comment cela fonctionne plus tard.

1. Prérequis

— Savoir programmer en PHP orienté objet et maîtriser les namespaces ;
— Savoir utiliser composer ;
— Connaître les normes de développement PSR (et les utiliser).

2. Pourquoi phpspec ?

Si on écoutait son créateur, il dirait probablement que phpspec n'est **pas** du test unitaire mais de la spécification.

En réalité lorsque l'on utilise phpspec on écrit ce que doit faire le code. On écrit donc sa spécification, mais cela permet également de le tester puisque phpspec va vérifier que le code fonctionne selon la spécification que nous lui fournissons.

En outre phpspec est orienté SDD, c'est à dire que dans l'idéal, il faudrait créer la spécification avant d'écrire notre code réel, et vous allez voir qu'on en tire un avantage.

3. Comment l'installer ?

phpspec s'installe facilement à l'aide de composer, c'est pour cela que j'ai demandé que vous sachiez l'utiliser.

3.1. Installation

Pour l'installer il nous suffira donc d'installer en l'ajoutant au fichier composer.json de votre projet :

```
$ composer require phpspec/phpspec
```

Votre fichier `composer.json` devrait ressembler à ceci :

```
{
    "require-dev": {
        "phpspec/phpspec": "^6.0"
    },
    "config": {
        "bin-dir": "bin"
    },
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

Rapide explication de ce que l'on fait :

- `require-dev` : on place la dépendance en dépendance de développement, inutile d'installer phpspec dans un environnement de production ;
- `config.bin-dir` : les fichiers binaires seront placés dans le dossier bin de votre dossier (si vous utilisez git, n'oubliez pas d'ajouter ce dossier à votre fichier .gitignore) ;
- `autoload` : On précise à composer comment charger vos classes (si vous utilisez un framework, cette option est probablement déjà configurée), ici on configure le dossier "src" comme dossier du code de notre projet, le namespace sera "App".

3.2. Vérifier que l'installation fonctionne

Tapez `bin/phpspec help`, cela devrait vous afficher l'aide de phpspec.

Si vous avez une erreur, réeffectuez les opérations que nous avons décrites.

4. Tester une classe

Tester un modèle simple n'a pas vraiment d'intérêt puisqu'il n'y a pas vraiment de logique dans le code. Cependant l'illustration est parlante. Prenons ces deux modèles :

```php
<?php

// src/Model/Article.php
namespace App\Model;

class Article
{
    private $id;
    private $title;
    private $content;
    private $username;

    public function getTitle()
    {
        return $this->title;
    }

    public function setTitle($title)
    {
        $this->title = $title;
        return $this;
    }

    public function setUsername(User $user)
    {
        $this->username = $user->getUsername();
    }

    public function getUsername()
    {
        return $this->username;
    }
}
```

Voici un exemple d'implémentation de classe `User`, nous ne la testerons pas mais vu que nous l'utilisons dans notre classe à tester, nous en avons tout de même besoin.

```php
<?php

// src/Model/User.php
namespace App\Model;

class User
{
    private $username;

    public function getUsername()
    {
        return $this->username;
    }
}
```

Si vous voulez simplement tester phpspec avec les exemples donnés dans ce cours vous pouvez enregistrer les fichiers sous les noms donnés dans les commentaires au début du code présenté ici.

4.1. Générer une spécification

Dans un premier temps, nous allons générer la spécification de cette classe à l'aide de l'exécutable phpspec. Pour cela utilisez la commande suivante :

```
$ bin/phpspec describe App/Model/Article
```

On utilise des "/" et pas des "\" car la ligne de commande les interprèterait d'une mauvaise façon. Il est tout à fait possible de spécifier le namespace en bonne et due forme en utilisant des guillemets : `"App\Model\Article"`

Si vous regardez à la racine de votre projet, vous devriez constater que phpspec a généré un dossier nommé « spec ». Ce dossier contient normalement les mêmes dossiers que votre « src » (ils représentent les namespaces).

Vous trouverez donc dans le namespace correspondant à votre classe, une classe nommée `VotreClasseSpec`. Pour l'exemple que nous allons donner, cela sera donc `ArticleSpec`.

Cette classe est donc la « spécification » de la votre, la seule chose que phpspec est capable de détecter automatiquement c'est que notre classe est bien une instance d'elle même... Pas très intéressant en somme (cela aurait pu être intéressant de tester que c'est bien une instance d'une interface donnée).

Finalement, il n'y a que deux choses importantes, la classe de spécification doit hériter de `ObjectBehavior`et son nom doit être composé de la classe qu'on souhaite tester et du suffix `Spec`.

4.2. Écrire nos tests

Nous allons écrire une méthode par comportement que nous souhaitons tester. Par convention on écrit ces comportements en anglais en utilisant le snake_case . Créons donc une méthode à notre objet qui va tester le comportement « enregistrer un titre » (car on doit pouvoir utiliser le setter de titre de l'objet).

```php
<?php

// spec/App/Model/ArticleSpec.php
namespace spec\App\Model;

use PhpSpec\ObjectBehavior;

class ArticleSpec extends ObjectBehavior
{
    function it_should_save_a_title()
    {
        $this->setTitle('Un titre au hasard');
        $this->getTitle()->shouldReturn('Un titre au hasard');
    }
}
```

La syntaxe peut paraître déroutante, on teste ce que retourne les méthodes de notre objet en appelant les méthodes comme si on était dans ce dernier. Cependant pour tester les valeurs de retour on peut utiliser des méthodes de test sur nos méthodes. phpspec fait une simulation, c'est comme si nous étions *dans* la classe que nous testons, on utilise donc `$this` pour exécuter les méthode sur cette dernière.

Décryptons un peu cela :

1. On appelle `$this->setTitle()`, phpspec comprend alors qu'on veut utiliser la méthode setTitle sur notre objet. Il s'exécute.
2. On veut vérifier que notre objet a bien enregistré le titre en appelant sa méthode getTitle, on utilise alors `$this->getTitle()`. Ici phpspec ne nous retournera pas le retour direct de la méthode que l'on souhaite appeler mais un objet spécial sur lequel on a quelques méthodes dont `shouldReturn()`, cette méthode permet en réalité d'écrire la spécification en prédisant ce que l'appel de notre méthode doit retourner.

Notez que la méthode `shouldReturn()` est un **matcher** dans le language de phpspec, vous pouvez trouver la liste des matchers sur la documentation de phpspec .

Le fait de dire que nous écrivons des prédictions n'est pas innocent car phpspec est basé sur une autre bibliothèque nommée prophecy.

> Vous remarquerez que l'on n'utilise pas le mot clé `public` pour définir la méthode, cela est une convention pour les spécifications phpspec. Cependant gardez à l'esprit que cela n'est valable que pour phpspec. Et personne ne vous interdit d'utiliser le mot clé `public`.

Finissons en lançant notre test avec la commande phpspec :

```
1 $ bin/phpspec run
```

Et voici à peu de choses près le rendu que vous devriez obtenir :

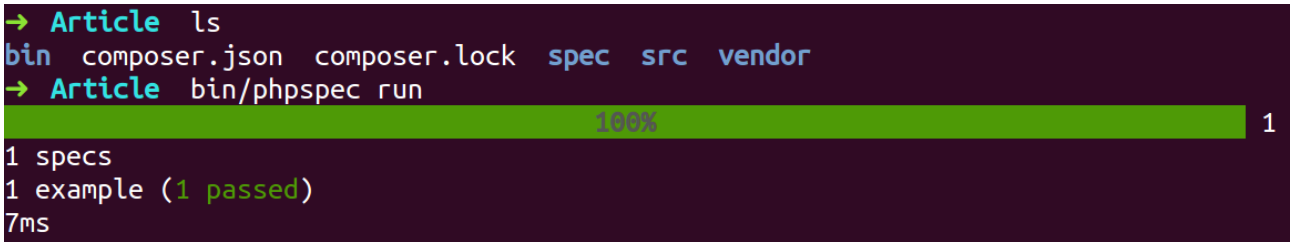

FIGURE 4.1. – Illustration phpspec en action

5. Les mocks avec phpspec

Les mocks sont des « faux » objets, en effet nos objets utilisent souvent d'autres objets. Avec phpspec tous les autres objets seront des *mocks* générés par le framework de test. Cela permet de tester notre objet dans un environnement clos et d'être sûr qu'un bug ne viendra jamais de notre code.

> Notez que cela pose un problème par rapport à l'application globale, nos tests ne garantiront pas que la globalité de notre application fonctionne. La plupart du temps quand on utilise phpspec on utilise un autre framework de tests qui permet de tester la globalité du site. Behat est un excellent complément à phpspec (qui plus est, du même créateur !).

Et donc comment créer nos fameux mocks ? De la façon la plus simple du monde : en réclamant des objets en paramètre à nos tests ! Testons nos méthodes d'enregistrement du nom d'utilisateur qui utilisent un objet user.

```php
<?php

// spec/MyApplication/Model/ArticleSpec.php
namespace spec\MyApplication\Model;

use PhpSpec\ObjectBehavior;
use App\Model\User;

class ArticleSpec extends ObjectBehavior
{
    function it_should_save_a_username_using_a_user(User $user)
    {
        $user->getUsername()->willReturn('Nek')->shouldBeCalled();
        $this->setUsername($user);
        $this->getUsername()->shouldReturn('Nek');
    }
}
```

Ce code devrait vous sembler simple à comprendre, dans le doute détaillons un peu ce que j'ai fait :

1. J'ai utilisé un objet de type `User` que j'ai réclamé à phpspec en le typant. Ce dernier va alors nous générer un **faux** objet de type User (ces objets sont appelés mocks), les classes que nous testerons n'y verront que du feu.
2. J'ai informé phpspec sur la valeur que devait retourner la méthode `getUsername()` en utilisant `willReturn()`.
3. J'ai également enchaîné avec la méthode `shouldBeCalled()`, cette méthode est totalement optionnelle mais elle va ajouter un test supplémentaire car elle va provoquer une erreur de notre test si la méthode `getUsername()` n'est pas appelée par notre objet testé.
4. Enfin, de la même façon que lors du test précédent nous utilisons les méthodes de notre objet pour les tester.

Les interfaces peuvent être utilisées comme type pour les mocks ! Et c'est également le cas pour les classes abstraites.

6. Préparer le test

Une subtilité à laquelle vous avez peut être pensé est la question du constructeur. Ok, ici nous n'avions pas encore de constructeur et donc nous pouvions tester nos méthodes tranquillement. Mais que se passerait-il si notre constructeur était utile et attendait des paramètres ?

Le test planterait.

La solution est relativement simple, il s'agit d'implémenter une méthode spécifique à phpspec : `let`. Cette dernière s'exécutera avant chaque test afin de préparer les mocks. Vous l'aurez deviné, ce nom de fonction n'est donc pas valable pour un test.

Imaginons le constructeur suivant :

```php
<?php

// Je ne réécris pas les use et namespaces

class Article
{
    public function __construct(User $user)
    {
        $this->user = $user;
    }
}
```

Ici nous allons avoir besoin de l'utilisateur dès l'initialisation. Voyons comment la méthode `let` peut s'utiliser dans notre cas :

```php
<?php

class ArticleSpec extends ObjectBehavior
{
    function let(User $user)
    {
        $this->beConstructedWith($user);
        $user->getUsername()->willReturn('Nek');
    }
}
```

Voici un petit récapitulatif de ce qui se passe ici pour ceux qui sont un peu perdus :

1. Nous déclarons la fonction `let` et spécifions en commentaires que l'objet attendu est un `User`, phpspec renverra un objet différent qui agira comme un `User` ;
2. On lui spécifie que notre objet doit être construit avec un utilisateur généré par PHPSpec ;
3. On spécifie à notre objet `User` qu'il doit retourner « Nek » à l'appel de la méthode `getUsername()`, de cette façon si la classe que nous testons appelle cette méthode de l'objet user, elle ne recevra pas `null`.

Vous l'avez peut-être deviné mais si on ajoute des tests et qu'on réutilise la variable `$user` en y spécifiant le type dans la PHPDoc et le nom dans la variable, on récupère la même instance, comme ça nous évite de devoir redéfinir à chaque test ce que l'objet doit retourner ;-) .

7. Vérifier le type de votre classe

Même si l'intérêt peut paraître étrange, si vous écrivez votre spécification **avant** de mettre en place le code correspondant, écrire le type peut vous aider à mieux concevoir votre application. Notamment parce que ce type n'est pas seulement le namespace (après tout vous avez besoin du namespace pour créer votre spec !) mais c'est également les interfaces et classes héritées.

phpspec nous permet de vérifier cela très simplement, considérons la classe suivante et sa spec :

```php
<?php

class User implements UserInterface
{
    // Une fois de plus je ne réécris pas tout ce qu'il peut y
        avoir autour
    public function getUsername()
    {
        return $this->username;
    }
}
```

```php
<?php

class UserSpec extends ObjectBehavior
{
    public function it_is_initializable()
    {
        $this-
            >shouldHaveType('Symfony\Component\Security\Core\User\UserInterface
    }
}
```

Ici c'est la fonction `it_is_initializable` qui va nous permettre de tester le type de notre classe. Vous pouvez bien entendu tester plusieurs type (une classe peut hériter d'une autre et avoir plusieurs interfaces).

Grâce à ce test vous vérifiez que votre classe "User" implémente bien l'interface `User` de Symfony. Et comme cette dernière est très importante pour votre code, si quelqu'un dans le futur la supprime vous aurez une erreur dans votre test !

8. Un exemple concret de classe à tester

Je prends cet exemple spécifique car il va nous permettre de voir plein de nouvelles choses que nous n'avons pas encore eu l'occasion d'utiliser dans phpspec. Cependant, vous verrez que cela semble assez naturel.

8.1. Énoncé

Nous voulons écrire une classe qui va nous générer une couleur aléatoire en hexadécimal.

8.2. Écriture d'une spec

Avant même d'écrire notre code, nous allons commencer par imaginer comment notre classe va fonctionner, nous allons créer sa classe de spécificité.

Pour cela nous avons besoin d'imaginer son namespace. Je propose d'imaginer que notre application est namespacée « App ». Nous avons également besoin d'un nom de namespace intermédiaire qui va indiquer la fonction de notre classe (ou son appartenance d'un point de vue domaine, c'est vous qui choisissez !). Il faudra ensuite lui trouver un nom.

Je propose les choses suivantes :

— Namespace `App\Utils` ;
— Nom de classe `Color`.

Nous avons donc au final le nom de classe complet suivant : `App\Utils\Color`

Tout ceci vous semble peut-être bête, mais dans la programmation avancée le nommage des choses est très important et a un impact très fort sur la maintenabilité de vos projets.

J'ai oublié comment faire pour créer les specs de base !

Pas de soucis ! Utilisons la commande de phpspec qui permet de créer des specs :

```
$ bin/phpspec describe "App\Utils\Color"
```

Et hop, phpspec nous a généré notre spec !

```
<?php

namespace spec\App\Utils;

use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class ColorSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType('App\Utils\Color');
```

```
13      }
14  }
```

Voici donc notre spec qui ne teste rien d'autre que vérifier le nom et namespace de notre classe finalement.

8.3. Imaginer le fonctionnement

Pour obtenir notre couleur nous allons devoir imaginer comment nous allons appeler une méthode sur notre classe.

Devant la simplicité de la chose je propose qu'on définisse une méthode `randomHexaColor` qui sera `static` sur notre classe.

Ajoutons donc notre test à notre spec :

```php
<?php
    function it_should_generate_a_color()
    {
        self::randomHexaColor()->shouldBeAnHexadecimalColor();
    }
```

Là **nous avons un problème**. Nous avons écrit un test qui vérifie bien la nature du résultat, cependant phpspec (ou plutôt prophecy) ne connaît pas cette méthode. Mais heureusement phpspec nous permet de la définir directement dans la spec ! Voici comment faire :

```php
<?php
    public function getMatchers()
    {
        return [
            'beAnHexadecimalColor' => function ($subject) {
                return (bool) preg_match('/#(?:[0-9a-fA-F]{6})/',
                    $subject);
            }
        ];
    }
```

Les méthodes de vérification sont appelées « matcher », on les définie en retournant un tableau dans la méthode `getMatchers`. Si elles retournent `true` alors la valeur est considérée comme correcte. Si ce n'est pas le cas elle est considérée comme fausse et le test n'est pas validé.

i Vous l'avez peut être remarqué, je n'utilise pas systématiquement le mot clé `public`, c'est une convention de phpspec. Tout ce qui sert à tester ne doit pas utiliser ce mot clé (PHP

considère ces méthodes publiques tout de même). En revanche tout ce qui peut être « moteur » doit utiliser le mot clé comme dans les conventions classiques de PHP. Mais tout ceci est uniquement **conventionnel**, vous faites ce que vous voulez.

8.4. Récapitulons

Notre test est à présent complet ! Il ne reste plus qu'à écrire le code correspondant.

Je vous donne le code complet de la spec au cas où vous vous seriez perdus en route :

```php
<?php

namespace spec\App\Utils;

use PhpSpec\ObjectBehavior;

class ColorSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType('App\Utils\Color');
    }

    function it_should_generate_a_color()
    {
        self::randomHexaColor()->shouldBeAnHexadecimalColor();
    }

    public function getMatchers()
    {
        return [
            'beAnHexadecimalColor' => function ($subject) {
                return (bool) preg_match('/#(?:[0-9a-fA-F]{6})/',
                    $subject);
            }
        ];
    }
}
```

8.5. Écrivons le code de notre classe

Parce que nous sommes des flemmards, phpspec est capable de nous générer notre classe ! Il suffit de lancer l'exécution des tests et lorsqu'il ne trouvera pas quelque chose qu'il doit tester il va nous proposer de l'ajouter automatiquement. Essayez par vous même :

```
$ bin/phpspec run
```

Il ne vous reste plus qu'à lui dire `Y` (pour « yes ») lorsque vous voulez qu'il créé les éléments pour vous.

Cependant il n'est pas capable de deviner le fonctionnement de notre classe (ça serait trop beau). Je vous donne donc directement un code qui va fonctionner avec notre spec. Cependant on aurait pu faire plusieurs types d'implémentation, j'ai utilisé deux fonctions car je trouvais cela plus simple à la lecture et cela phpspec ne pouvait pas le deviner.

```php
<?php

namespace App\Utils;

class Color
{
    public static function randomHexaColor()
    {
        return '#' . self::randomHexaPart() .
            self::randomHexaPart() . self::randomHexaPart();
    }

    private static function randomHexaPart()
    {
        return str_pad( dechex( mt_rand( 0, 255 ) ), 2, '0',
            STR_PAD_LEFT);
    }
}
```

Une fois notre classe complète, on relance les tests pour vérifier que tout fonctionne bien :

```
$ bin/phpspec run
```

Vous avez la théorie, mais sachez qu'en pratique vous allez être amené à voir des cas que nous n'avons pas présenté. La documentation de phpspec ne vous aidera pas beaucoup à ce sujet, vous devriez quand même la garder dans un coin en cas de trou de mémoire ! http://www.phpspec.net/

Cependant phpspec étant basé sur prophecy, je vous encourage grandement à vous référer à la documentation de ce dernier disponible ici : https://github.com/phpspec/prophecy#prophecy

9. Remerciements

— Garfieldfr pour sa revue technique ;
— albert733 pour la revue orthographique ;
— L'équipe de Zeste De Savoir qui fait un taff excellent avec ce site :-) .