Integração PHPspec com Laravel

Para realizar teste de integração com o phpspec, muitas vezes precisaremos acessar os serviços do laravel. Neste caso iremos nos deparar com problemas, pois, diferente do PHPUnit, o PHPSpec não é nativo do Laravel, e portanto, não executa “dentro” do ambiente de uma aplicação Laravel.

Em outras palavras, ao tentar acessar uma fachada do Artisan, Factory ou outros, iremos nos deparar com o erros do tipo: Classe não encontrada.

Para contornar este problema, é necessário realizar algumas modificações no projeto.

1. Instalação da extensão phpspec-laravel
composer require --dev "benconstable/phpspec-laravel:~4.0"

2. Configuração do phpspec-laravel no arquivo phpspec.yml, incluindo as seguintes linhas:

```
extensions:
    PhpSpec\Laravel\Extension\LaravelExtension:
        testing_environment: "testing"
```

3. Modificação no arquivo de teste, que deve incluir a classe LaravelObjectBehavior
use PhpSpec\Laravel\LaravelObjectBehavior;

e modificar o “extend” de sua classe de ObjectBehavior para LaravelObjectBehavior

Exemplos de Uso

Caso 1: Teste Unitário do Model

Obs: Para realizar a validação dos campos, utilizaremos a classe Validator do Laravel.

Regras:

- O nome do funcionário é obrigatório
- O apelido do funcionário é obrigatório

1. No Model Funcionário, adicionaremos as regras de validação e as mensagens de erro

```
class Funcionario extends Model
{
    protected $fillable = ['nome', 'apelido'];
    public static $rules = [
      'departamento' => 'required',
      'nome' => 'required|min:10',
      'apelido' => 'required'
    ];

    public static $messages = [
      'required' => 'O campo :attribute é obrigatório',
      'nome.min' => 'O campo nome deve ter ao menos 10 letras',
    ];
```

As mensagens de erro podem ser genéricas, ou seja, aplicadas a todos os atributos (a primeira: required) ou podem ser específicas para cada atributo (a segunda: nome.min). Ao trabalhar com regras genéricas, podemos ter acesso ao nome do atributo que gerou o erro com o :attribute

2. Em seguida, criaremos o a classe de teste do FuncionarioValidator
Obs. A classe FuncinoarioValidator ainda não foi criada, seguindo o método TDD, o teste é feito antes da classe.

vendor/bin/phpspec OrgTabajara/Validator/FuncionarioValidator
vendor/bin/phpspec run

O segundo comando irá perguntar se deseja criar a classe FuncionarioValidator, responda sim (y).

Na classe de teste FuncinoarioValidatorSpec, deveremos criar os métodos para testar as regras. Cada método deve iniciar com a palavra it ou its, seguindo o padrão de escrita snake case:

```
    function it_o_nome_do_funcionario_eh_obrigatorio() {
         $funcionario = new Funcionario;
         $funcionario->nome = "";
         $funcionario->apelido = "igor";
         $funcionario->departamento = 1;

         $this->shouldThrow('OrgTabajara\Validator\ValidationException')
              ->duringValidate($funcionario->toArray());
    }

    function it_o_apelido_do_funcionario_eh_obrigatorio() {
         $funcionario = new Funcionario;
         $funcionario->nome = "Igor Medeiros Vanderlei";
         $funcionario->apelido = "";
         $funcionario->departamento = 1;

         $this->shouldThrow('OrgTabajara\Validator\ValidationException')
              ->duringValidate($funcionario->toArray());
    }
```

Obs: Cada função deve testar apenas uma regra, que deve isolar uma funcionalidade.

3. Execute novamente o comando (vendor/bin/phpspec run), que deverá perguntar se você deseja criar o método validate na classe FuncionarioValidator, responda sim.

4. Edite a classe FuncionarioValidator, para implementar o corpo do método validate:

```
namespace OrgTabajara\Validator;
use OrgTabajara\Funcionario;

class FuncionarioValidator
{
    public static function validate($dados)
    {
        $validator = \Validator::make($dados,
                                      Funcionario::$rules,
                                      Funcionario::$messages);
        if(!$validator->errors()->isEmpty())
         throw new ValidationException($validator, "Erro ao validar o funcionario"
    }
}
```

A classe ValidationException foi criada na mesma pasta do Validator

```php
<?php

namespace OrgTabajara\Validator;

class ValidationException extends \Exception
{
   protected $validator;

    public function __construct($validator, $text = 'TODO: write validator text')
    {
        parent::__construct($text);
        $this->validator = $validator;
    }

    public function getValidator() {
      return $this->validator;
    }
}
```

Observe que estamos utilizando um serviço específico do Laravel (o Validator), portanto a classe de teste (FuncinoarioValidatorSpec) deve ter a modificação explicada na página 1, ficando:

```php
namespace spec\OrgTabajara\Validator;

use OrgTabajara\Validator\FuncionarioValidator;
use PhpSpec\ObjectBehavior;
use Prophecy\Argument;
use OrgTabajara\Funcionario;
use PhpSpec\Laravel\LaravelObjectBehavior;

class FuncionarioValidatorSpec extends LaravelObjectBehavior
{
```

Execute o teste novamente (vendor/bin/phpspec run), que agora deverá passar.

Importante!!! Não é preciso realizar um teste exaustivo em cada regra de validação, pois, como estamos utilizando o sistema de validação do Laravel, podemos considerar que este já foi amplamente testado pela sua equipe de desenvolvimento.

Caso 2: Teste com Banco de Dados

Regra:

- Um funcionário não pode enviar uma mensagem para si mesmo.

Neste caso, a implementação do teste tem uma dependência com a classe funcionário, que está armazenado no banco de dados.

Para esta situação, é desejável que o teste seja realizado em um banco de dados cujo estado é conhecido. Entretanto, os casos de teste podem modificar o banco, e cada caso de teste deve ser independente da execução dos demais. Assim, é preferível que a cada execução de um caso de teste, tenhamos uma versão intacta do banco de dados no estado conhecido.

A solução imediata poderia ser executar o comando “migrate:refresh –seed” antes de realizar cada caso de teste. Apesar de funcional, esta solução quando aplicada ao sgbd de desenvolvimento (postgresql, mysql, etc) pode acarretar em um teste mais lento do que o desejado.

Diante do exposto, realizaremos os testes o banco de dados sqlite em memória, como segue:

1. Crie o arquivo .env.testing e neste arquivo coloque as configurações do sqlite em memória:

```
DB_CONNECTION=sqlite
DB_DATABASE=:memory:
```

Obs. O **.testing** veio da configuração feita no phpspec.yml, no passo 2 da primeira página:

extensions:
  PhpSpec\Laravel\Extension\LaravelExtension:
    testing_environment: **"testing"**

2. Crie a classe de teste do MensagemValidator
vendor/bin/phpspec OrgTabajara/Validator/MensagemValidator
vendor/bin/phpspec run

3. Adicione o método let, para inicializar o banco e acrescente o teste:

```
function let() {
   \Artisan::call("migrate");
   \Artisan::call("db:seed");
}

function its_funcionario_nao_pode_enviar_uma_mensagem_para_si_mesmo() {
  $mensagem = factory(\OrgTabajara\Mensagem::class)->make();
  $mensagem->destinatario()->associate($mensagem->remetente);

  $this->shouldThrow('OrgTabajara\Validator\ValidationException')
     ->duringValidate($mensagem->toArray());
}
```

Observe o uso da factory para reduzir o trabalho.

4. Modifique a classe MensagemValidatorException para utilizar a extensão phpspec-laravel

5. Os outros testes podem ser feitos normalmente utilizando o factory:

```
function its_texto_deve_ter_o_minimo_de_caracteres(){
  $mensagem = new Mensagem;
  $mensagem = factory(\OrgTabajara\Mensagem::class)->make();
  $mensagem->texto = "teste";

  $this->shouldThrow('OrgTabajara\Validator\ValidationException')
     ->duringValidate($mensagem->toArray());
}
```

Obs. A factory foi modificada para garantir que uma mensagem não tenha o remetente igual ao destinatário, pois isso poderia influenciar neste segundo teste. De uma forma geral, devemos garantir que a factory vai construir o objeto em um estado consistente.

6. Execute o phpspec para criar a classe MensagemValidator e o método validate.

7. Acrescente as regras de validação no model Mensagem:

```
class Mensagem extends Model
{
    protected $fillable = ['titulo', 'texto'];
    public static $rules = [
      'titulo' => 'required',
      'texto' => 'required|min:10|max:200',
    ];
    public static $message = [
      'required' => 'O campo :attribute é obrigatório'
    ];
```

8. Implemente o método validate na classe MensagemValidator

```php
<?php

namespace OrgTabajara\Validator;
use OrgTabajara\Mensagem;
use OrgTabajara\Validator\ValidationException;
use Illuminate\Support\Facades\Validator;

class MensagemValidator
{
    public static function validate($dados)
    {
      $validator = Validator::make($dados,
                                   Mensagem::$rules,
                                   Mensagem::$message);
      if($dados['remetente']['id'] == $dados['destinatario']['id']) {
         $validator->errors()->add('destinatario',
                                   'O destinatario deve ser diferente do remetente');
      }
      if(!$validator->errors()->isEmpty())
         throw new ValidationException($validator, "Erro ao validar a mensagem");
     }
}
```

Observe que a implementação da regra “remetente diferente do destinatário” não foi feita com as regras de validação do Laravel, entretanto, o erro resultante desta verificação deve passar para o objeto $validator, de modo que tenhamos um único ponto de acesso às mensagens de erro.
O método “$validator→errors()→add” recebe como parâmetros o campo e a mensagem, nesta ordem.

Testes com PHPUnit

Caso 1: Teste Unitário do Model

Utilizaremos os mesmo model e validator utilizado no Caso 1 do phpspec, que neste ponto já estão implementados.

1. Criar a classe de teste do PHPUnit

php artisan make:test FuncionarioTest –unit

2. Editar a classe FuncionarioTest (localizada no diretório tests/Unit), incluindo uma função para cada teste. Cada nome de função deve iniciar com a palavra test e utilizar o padrão de escrita camelCase.

```
class FuncionarioTest extends TestCase
{

    /**
    * @expectedException OrgTabajara\Validator\ValidationException
    */
    public function testeFuncionarioNomeInvalido() {
        $funcionario = new Funcionario;
        $funcionario->departamento = 1;
        $funcionario->nome = "abf";
        $funcionario->apelido = "abf";
        FuncionarioValidator::validate($funcionario->toArray());
    }
}
```

3. Executar o teste
vendor/bin/phpunit

ou

vendor/bin/phpunit UnitTest tests/Unit/FuncionarioTest.php (caso existam várias classes de teste, mas você deseja executar apenas uma)

Caso 2: Teste com Banco de Dados

Para fazer com que o phpunit trabalhe com o banco de dados sqlite em memória, edite o arquivo phpunit.xml e inclua a variável de configuração DB_DRIVER e DB_DATABASE com os valores sqlite e :memory: , respectivamente.

```
    <php>
        <env name="APP_ENV" value="testing"/>
        <env name="BCRYPT_ROUNDS" value="4"/>
        <env name="CACHE_DRIVER" value="array"/>
        <env name="SESSION_DRIVER" value="array"/>
        <env name="QUEUE_DRIVER" value="sync"/>
        <env name="MAIL_DRIVER" value="array"/>
        <env name="DB_DRIVER" value="sqlite"/>
        <env name="DB_DATABASE" value=":memory:"/>
    </php>
</phpunit>
```

1. Agora modifique o arquivo tests/CreatesApplication.php para incluir os comandos migrate e db:seed

```
use Illuminate\Contracts\Console\Kernel;

trait CreatesApplication
{

    public function setUp() {
        parent::setUp();
        \Artisan::call('migrate:refresh');
        \Artisan::call('db:seed');
    }

    public function tearDown() {
        \Artisan::call('migrate:rollback');
        parent::tearDown();
    }
```

2. Agora podemos criar os casos de teste que utiliza dados do banco.
php artisan make:test MensagemTest

```
    /**
     * @expectedException OrgTabajara\Validator\ValidationException
     */
    public function testeRemetenteIgualDestinatario() {
        $mensagem = factory(\OrgTabajara\Mensagem::class)->make();
        $mensagem->destinatario()->associate($mensagem->remetente);
        MensagemValidator::validate($mensagem->toArray());
    }
```

Outros Testes com Banco

Verificar se uma mensagem enviada para um destinatário irá aparecer em sua caixa de entrada

1. Incialmente irei criar a função enviarMensagem no model Funcionario.

```
public function enviarMensagemPara($funcionario, $titulo, $texto){
  $mensagem = new Mensagem;
  $mensagem->titulo = $titulo;
  $mensagem->texto = $texto;
  $mensagem->destinatario()->associate($funcionario);
  $this->caixasaida()->save($mensagem);
}
```

---

2. Agora vou incluir a função de teste na classe MensagemTest

```
public function testeEnviarMensagemCaixaEntradaDestinatario() {
   $f1 = Funcionario::find(1);
   $f2 = Funcionario::find(2);
   $qtd_entrada_f2 = $f2->caixaentrada->count();

   $f1->enviarMensagemPara($f2, "titulo", "texto");

   $f2 = Funcionario::find(2);
   $qtd_entrada_f2_2 = $f2->caixaentrada->count();

   $this->assertEquals($qtd_entrada_f2_2, $qtd_entrada_f2 + 1);
}
```

Verificar se uma mensagem enviada na caixa de saída do remetente

```
public function testeEnviarMensagemCaixaSaidaRemetente() {
   $f1 = Funcionario::find(1);
   $f2 = Funcionario::find(2);
   $qtd_saida_f1 = $f1->caixasaida->count();

   $f1->enviarMensagemPara($f2, "titulo", "texto");
   $f1 = Funcionario::find(1);
   $qtd_saida_f1_2 = $f1->caixasaida->count();

   $this->assertEquals($qtd_saida_f1_2, $qtd_saida_f1 + 1);
}
```

Caso 3: Teste de Controller

O que deve ser testado no controller?
1. Garantir que os métodos que manipulam o banco de dados estão sendo chamados de forma correta;
2. Verificar se as respostas e redirecionamentos estão corretos;
3. Verificar se as variáveis corretas estão sendo enviadas para as views.

O que não testar no controller?
Todo o resto, por exemplo, verificar se o dado está sendo armazenado no banco não é função do teste de controller.

Três Regras para testar o controller:
- **Isolate:** Utilizar objetos Mock para todas as dependências
- **Call:** Chamar o método desejado
- **Ensure:** Realiar enunciados, verificando que tudo foi "setado" adequadamente

Refatorando o controller para facilitar o teste com isolamento.

Em vez de instanciar os objetos que serão utilizados no controller (com o new), devemos passá-lo como parâmetro do construtor, para fazer com que o IoC do Laravel inicialize esses objetos:

```
class FuncionarioController extends Controller
{
    protected $funcionario;

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

Após a refatoração, o método deve ficar desta forma:

```
    public function store(Request $request) {
        try {
            FuncionarioValidator::validate($request->all());
            $this->funcionario->fill($request->all());
            $this->funcionario->save();
            return "ok";
        } catch(ValidationException $e) {
            return "exception";
        }
    }
```

Obs: O return "ok"; e o return "exception" poderão ser alterados posteriormente.

Antes de testar o controler, devemos lembrar de incluir a rota:

```
Route::post('funcionario/store', 'FuncionarioController@store');
```

Para criar a classe que testa o controller execute

php artisan make:teste FuncionarioControllerTest

(sem a opção –unit) Neste caso, a classe de teste será criada no diretório /tests/Feature

Acrescentando os métodos:

```
public function testeCadastrarComDadosCorretos()
{
  $f = new Funcionario([
        'nome' => "Igor Medeiros",
        'apelido' => "Igor"
  ]);
  $f->departamento = 1;

  $funcionarioMock = \Mockery::mock('OrgTabajara\Funcionario');
  $funcionarioMock->shouldReceive('save')->once();
  $funcionarioMock->shouldReceive('fill')->once();

  $this->app->instance('OrgTabajara\Funcionario', $funcionarioMock);

  $response = $this->call('POST', 'funcionario/store', $f->toarray());
  $response->assertSee("ok");
}
```

Observe o uso do Mockery, para isolar o teste do controller das funções de banco. No controller, só estamos interessados em saber se os dados estão chegando corretamente, se as funções estão sendo chamadas corretamente e se os direcionamentos estão corretos.

Neste caso de teste existem dois caminhos, um com os dados corretos (exemplo acima) e o outro com os dados incorretos.

No segundo teste, devemos garantir que o método save não foi executado. Desta forma estará testado que o o controller, com o auxílio do validator, escolheu o caminho correto, dependendo dos dados passados no teste.

```php
public function testeCadastrarComErro() {
  $f = new Funcionario([
        'nome' => "",
        'apelido' => "Igor"
  ]);
  $f->departamento = 1;

  $funcionarioMock = \Mockery::mock('OrgTabajara\Funcionario');
  $funcionarioMock->shouldNotReceive('save');
  $this->app->instance('OrgTabajara\Funcionario', $funcionarioMock);

  $response = $this->call('POST', 'funcionario/store', $f->toarray());
  $response->assertSee("exception");
}
```