Jeff Simons Decena Follow
May 29, 2017 · 3 min read

Repository pattern with Laravel

Last time, I wrote an article to get your feet wet on Test Driven Development (TDD). You may refer to this post: Simple TDD in Laravel with 11 steps

Today, let us refactor what we have accomplished on that post to make it more manageable. We will use the **REPOSITORY** pattern. With this way, methods are reusable in the other parts of the application by doing Dependency Injection (DI).

Fork my TDD e-commerce application: https://github.com/jsdecena/laracom (check out the `/tests` folder)

Edit: I created a base repository package that can speed up your development :)

Let's review the test we have created.

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ArticleTest extends TestCase
{
    /** @test */
    public function it_can_create_an_article()
    {
        $title = $this->faker->word;

        $data = [
            'title' => $title,
            'slug' => str_slug($title),
            'excerpt' => $this->faker->sentence,
            'content' => $this->faker->paragraph,
            'author_id' => 1,
```

And also check the output of the `phpunit` command on the terminal to see if it still passes.

```
➜  intranet git:(develop) ✗ phpunit
PHPUnit 5.7.20 by Sebastian Bergmann and contributors.

.
1 / 1 (100%)

Time: 401 ms, Memory: 16.00MB

OK (1 test, 9 assertions)
```

Good! It is passing!

Let’s review the controller that makes the test pass as well.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Articles\Requests\CreateArticleRequest;
use App\Http\Controllers\Controller;

class ArticlesApiController extends Controller
{
    /**
     * Store all the data request in the database
     *
     * @param CreateArticleRequest $request
     * @return \Illuminate\Http\JsonResponse
```

But there might be implication on this type of implementation. When we want to create an article via the command line (or anywhere), we have to recreate what we have done in the controller to achieve the same result right? Is that good? Nope. That is not **DRY** (Don't Repeat Yourself) at all.

So what can we do?

There are many approaches on every scenario but I would like to take the **REPOSITORY** pattern approach.

. . .

REPOSITORY

There are also different approaches in using the repository pattern but we will use the one that uses the Eloquent ORM.

So how do we refactor the controller's block of code? See below.

```php
<?php

namespace App\Http\Controllers\Api;

use App\Articles\Repositories\Interfaces\ArticleRepository
use App\Articles\Requests\CreateArticleRequest;
use App\Http\Controllers\Controller;

class ArticlesApiController extends Controller
{
    private $articleRepo;

    public function __construct(ArticleRepositoryInterface
    {
        $this->articleRepo = $articleRepository;
    }

    /**
     * Store all the data request in the database
```

So what happened to this? What voodo has happened? If we run again `phpunit`, it will turn to RED and fail.

Let me explain what happened. We moved our implementation of

```
$article = Article::create($request->all());
```

to a repository class that has a **create** method.

```
$article = $this->articleRepo->create($request->all());
```

Doing this, we will not be repeating again the code implementation on this **store** method in case we want to create an article. If we want to create an article, we just need to call the **create** method on the article repository class.

Maybe you are asking, you keep on saying the **ArticleRepository** class but you injected an **ArticleRepositoryInterface**, are those same?

Nope. But they go **together** to make the application more reliable. The interface is implented by a class to strictly follow the methods declared in there. This means, if a method is declared in the interface and not implemented by the class, the application will throw an error. Also, this will make the application more manageable in the future.

Laravel has DI mechanism as described on this part of the documentation.

How can we do all the things I am talking about?

Follow it with these simple steps:

STEP 1: CREATE THE INTERFACE AND ADD THE METHOD TO BE IMPLEMENTED

```
1  <?php
2
3  namespace App\Articles\Repositories\Interfaces;
4
5  use App\Base\BaseRepositoryInterface;
6
7  interface ArticleRepositoryInterface extends BaseRepositor
```

STEP 2: CREATE THE REPOSITORY CLASS

```php
<?php

namespace App\Articles\Repositories;

use App\Articles\Article;
use App\Articles\Repositories\Interfaces\ArticleRepository
use App\Base\BaseRepository;

class ArticlesRepository extends BaseRepository implements
{
    protected $model;

    /**
     * ArticlesRepository constructor.
     * @param Article $article
     */
    public function __construct(Article $article)
    {
        $this->model = $article;
    }
```

As you can see, we are extending a **BaseRepository** class. So, what does it do? Check below.

```php
<?php

namespace App\Base;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Illuminate\Pagination\LengthAwarePaginator;

abstract class BaseRepository
{
    protected $model;

    /**
     * BaseRepository constructor.
```

STEP 3: CREATE A REPOSITORY SERVICE PROVIDER

```php
<?php

namespace App\Providers;

use App\Articles\Repositories\ArticlesRepository;
use App\Articles\Repositories\Interfaces\ArticleRepository
use Illuminate\Support\ServiceProvider;

class RepositoryServiceProvider extends ServiceProvider
{
    /**
     * Bind the interface to an implementation repository
     */
    public function register()
```

Why do we need this? This tells the application which class implements the interface.

STEP 4: LET LARAVEL KNOW YOUR SERVICE PROVIDER

Add this into your `config/app.php` file.

```php
<?php

'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    ...
```

STEP 5: RUN YOUR PHPUNIT AND HOPE FOR THE BEST AGAIN

```
➜  intranet git:(develop) ✗ phpunit
PHPUnit 5.7.20 by Sebastian Bergmann and contributors.

.
1 / 1 (100%)

Time: 478 ms, Memory: 16.00MB
```

```
OK (1 test, 9 assertions)
```

Hooray! You are now a certified TDD-ier!

With all these in place, you can populate already your test file with other test scenarios like negative testing. Check if the application catches the errors it should throw. Errors like the user is not found should be caught. Negative testing might be covered in the next article.

So keep in mind the TDD cycle - RED, GREEN, REFACTOR.

/jsd