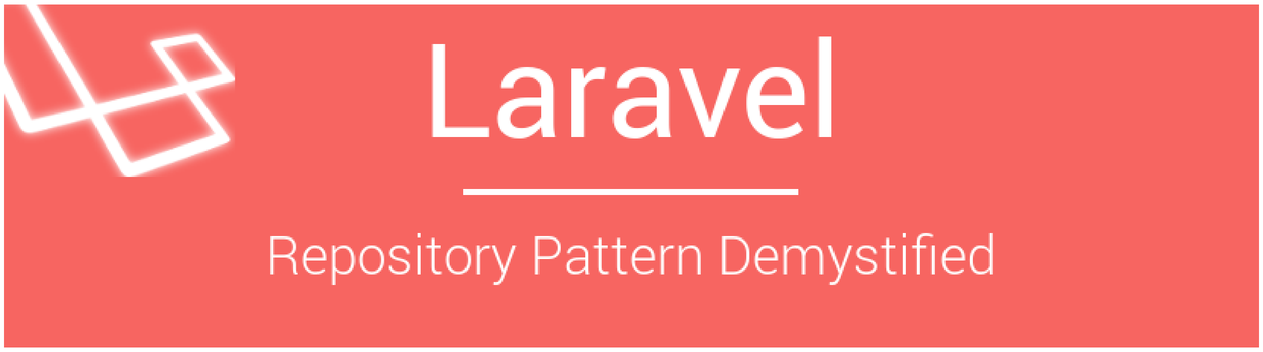
**Bosnadev – Code Factory**

*"You must unlearn what you have learned"*



# Using Repository Pattern in Laravel 5

These days there is a lot of buzz about software design patterns, and one of the most frequently asked questions is "How can I use *some pattern* with *some technology*". In the case of Laravel and the Repository pattern, I see often questions like ***"How I can use repository pattern in Laravel 4"*** or nowadays ***"..in Laravel 5"***. Important thing you must remember is that **design patterns do not depend on specific technology, framework or programming language**.

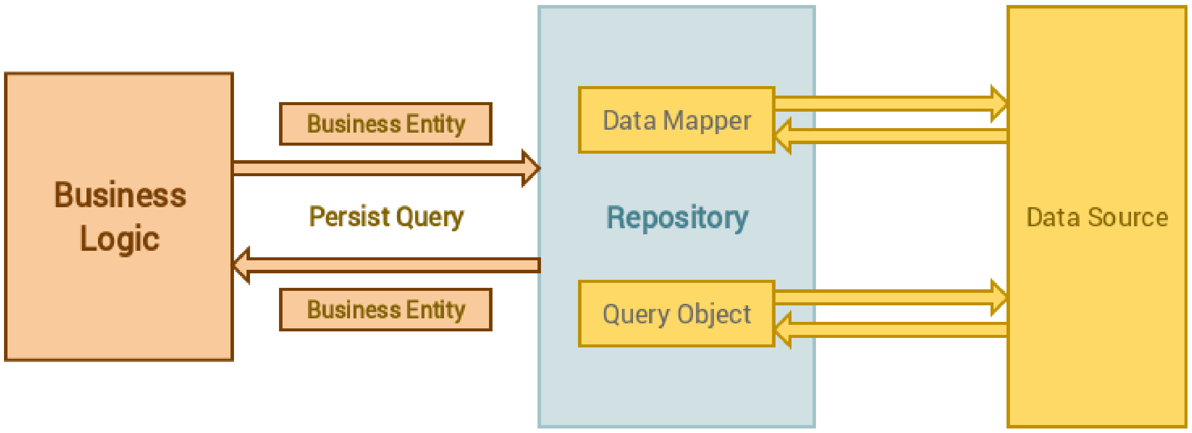
**Contents** [hide]

## Introduction

If you have really understood Repository Pattern then it does not matter what framework or programming language you are going to use. What is important is that you understand the **principle** behind the **Repository pattern**. Then you can implement it in whatever technology you want. With that in mind, let's start with the definition of the Repository pattern:

> *A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes.*

**Repository pattern** separates the data access logic and maps it to the business entities in the business logic. Communication between the data access logic and the business logic is done through interfaces.



To put it simply, **Repository pattern** is a kind of container where data access logic is stored. It hides the details of data access logic from business logic. In other words, we allow business logic to access the data object without having knowledge of underlying data access architecture.

The separation of data access from business logic have many benefits. Some of them are:

- Centralization of the data access logic makes code easier to maintain
- Business and data access logic can be tested separately
- Reduces duplication of code
- A lower chance for making programming errors

## It's all about interfaces

**Repository pattern** is all about interfaces. An interface acts like a contract which specify what an concrete class must implement. Let's think a little bit. If we have two data objects ***Actor*** and ***Film***, what are common set of operations that can be applied to these two data objects? In most situations we want to have the following operations:

1. Get all records
2. Get paginated set of records

3. Create a new record
4. Get record by it's primary key
5. Get record by some other attribute
6. Update a record
7. Delete a record

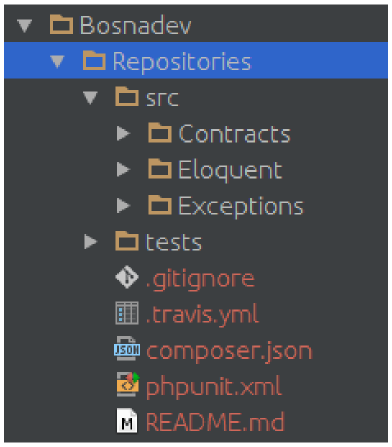
Can you see now how much duplicated code would we have if we implement this for each data object? Sure, for small projects it's not a big problem, but for large scale applications it's a bad news.

Now when we have defined common operations, we can create an interface:

```
interface RepositoryInterface {

    public function all($columns = array('*'));

    public function paginate($perPage = 15, $columns = array('*'));

    public function create(array $data);

    public function update(array $data, $id);

    public function delete($id);

    public function find($id, $columns = array('*'));

    public function findBy($field, $value, $columns = array('*'));
}
```

## Directory structure

Before we continue with creating concrete repository class that will implement this interface, let's think a bit how we want to organise our code. Usually, when  I create something, I like to think component way since I want to be able to reuse that code in other projects. My simple directory structure for the repositories component looks like this:



```
▼ Bosnadev
  ▼ Repositories
    ▼ src
      ▶ Contracts
      ▶ Eloquent
      ▶ Exceptions
    ▶ tests
      .gitignore
      .travis.yml
      composer.json
      phpunit.xml
      README.md
```

But it can be different, for example if component have configuration options, or migrations, etc.

Inside `src` directory I have three other directories: **Contracts**, **Eloquent** and **Exceptions**. As you can see, the folder names are pretty convenient for what we want to put there. In **Contracts** folder we put interfaces, or contracts as we call them earlier. **Eloquent** folder contains abstract and concrete repository class that implements contract. In **Exceptions** folder we put exceptions classes.

Since we are creating a package we need to create `composer.json` file where we define a mapping for namespaces to specific directories, package dependencies and other package metadata. Here is the content of `composer.json` for this package:

```json
{
  "name": "bosnadev/repositories",
  "description": "Laravel Repositories",
  "keywords": [
    "laravel",
    "repository",
    "repositories",
    "eloquent",
    "database"
  ],
  "licence": "MIT",
  "authors": [
    {
      "name": "Mirza Pasic",
      "email": "mirza.pasic@edu.fit.ba"
    }
  ],
  "require": {
    "php": ">=5.4.0",
    "illuminate/support": "5.*",
    "illuminate/database": "5.*"
  },
  "autoload": {
    "psr-4": {
      "Bosnadev\\Repositories\\": "src/"
    }
  },
  "autoload-dev": {
    "psr-4": {
      "Bosnadev\\Tests\\Repositories\\": "tests/"
    }
  },
  "extra": {
    "branch-alias": {
      "dev-master": "0.x-dev"
    }
  },
  "minimum-stability": "dev",
  "prefer-stable": true
}
```

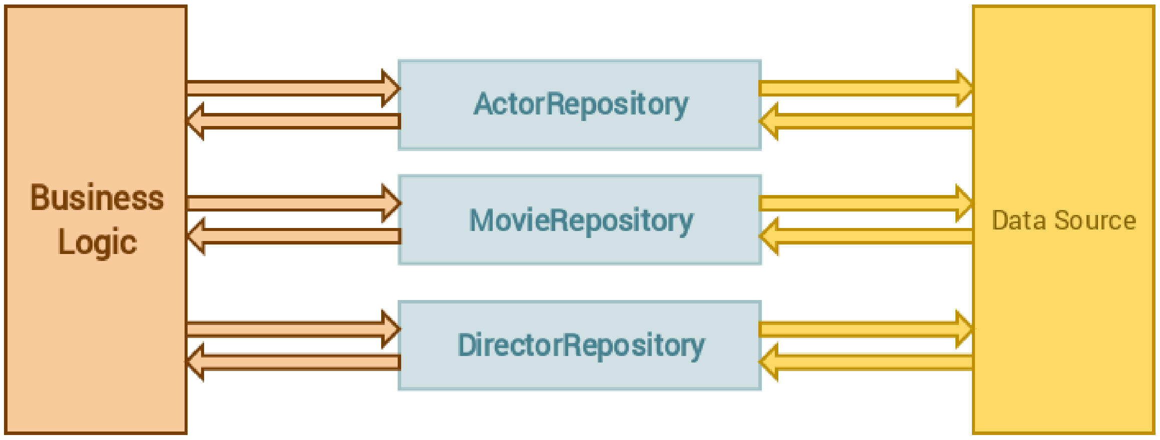
As you can see, we mapped namespace `Bosnadev\Repository` to the `src` directory. Another thing, before we start to implement *RepositoryInterface,* since it is located in the **Contracts** folder, we need to set correct namespace for it:

```php
<?php namespace Bosnadev\Repositories\Contracts;

interface RepositoryInterface {

...

}
```

We are now ready to start with the implementation of this contract.

## A Repository Implementation

Using repositories enables us to query the data source for the data, map the data to a business entity and persist changes in the business entity to the data source:



Of course, each concrete child repository should extend our abstract repository, which implements *RepositoryInterface* contract. Now, how would you implement this contract? Take a look at first method. What can you tell about it just by looking at it?

First method in our contract is conveniently named `all()` . It's duty is to fetch all records for the concrete entity. It accepts only one parameter `$columns` which must be an array. This parameter is used, as its name suggests, to specify what columns we want to fetch from the data source, and by default  we fetch them all.

For specific entity, this method could look like this:

```
public function all($columns = array('*')) {
    return Bosnadev\Models\Actor::get($columns);
}
```

But we want to make it generic, so we can use it wherever we want:

```
public function all($columns = array('*')) {
    return $this->model->get($columns);
}
```

In this case `$this->model` is an instance of `Bosnadev\Models\Actor` . Thus, somewhere in the repository we need to create a new instance of the given model. Here is one solution how you can implement this:

```
<?php namespace Bosnadev\Repositories\Eloquent;

use Bosnadev\Repositories\Contracts\RepositoryInterface;
use Bosnadev\Repositories\Exceptions\RepositoryException;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Container\Container as App;

/**
 * Class Repository
 * @package Bosnadev\Repositories\Eloquent
 */
abstract class Repository implements RepositoryInterface {

    /**
     * @var App
     */
    private $app;

    /**
     * @var
     */
    protected $model;

    /**
     * @param App $app
     * @throws \Bosnadev\Repositories\Exceptions\RepositoryException
     */
    public function __construct(App $app) {
        $this->app = $app;
        $this->makeModel();
    }

    /**
     * Specify Model class name
     *
     * @return mixed
     */
    abstract function model();

    /**
     * @return Model
     * @throws RepositoryException
     */
    public function makeModel() {
        $model = $this->app->make($this->model());

        if (!$model instanceof Model)
            throw new RepositoryException("Class {$this->model()} must be an instance of Illuminate\\D

        return $this->model = $model;
    }
}
```

Since we declared class as abstract, it means it must be extended by concrete child class. By declaring `model()` method as abstract we force the user to implement this method in the concrete child class. For example:

```
<?php namespace App\Repositories;

use Bosnadev\Repositories\Contracts\RepositoryInterface;
use Bosnadev\Repositories\Eloquent\Repository;

class ActorRepository extends Repository {

    /**
     * Specify Model class name
     *
     * @return mixed
     */
    function model()
    {
        return 'Bosnadev\Models\Actor';
    }
}
```

Now we can implement the rest of the contract methods:

```
<?php namespace Bosnadev\Repositories\Eloquent;

use Bosnadev\Repositories\Contracts\RepositoryInterface;
use Bosnadev\Repositories\Exceptions\RepositoryException;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Container\Container as App;

/**
 * Class Repository
 * @package Bosnadev\Repositories\Eloquent
 */
abstract class Repository implements RepositoryInterface {

    /**
     * @var App
     */
    private $app;

    /**
     * @var
     */
    protected $model;

    /**
     * @param App $app
     * @throws \Bosnadev\Repositories\Exceptions\RepositoryException
     */
    public function __construct(App $app) {
        $this->app = $app;
        $this->makeModel();
    }

    /**
     * Specify Model class name
     *
     * @return mixed
     */
    abstract function model();

    /**
     * @param array $columns
     * @return mixed
     */
    public function all($columns = array('*')) {
        return $this->model->get($columns);
    }

    /**
     * @param int $perPage
     * @param array $columns
     * @return mixed
     */
    public function paginate($perPage = 15, $columns = array('*')) {
        return $this->model->paginate($perPage, $columns);
    }

    /**
     * @param array $data
     * @return mixed
     */
    public function create(array $data) {
        return $this->model->create($data);
    }

    /**
     * @param array $data
     * @param $id
     * @param string $attribute
     * @return mixed
     */
    public function update(array $data, $id, $attribute="id") {
        return $this->model->where($attribute, '=', $id)->update($data);
    }

    /**
     * @param $id
     * @return mixed
     */
    public function delete($id) {
        return $this->model->destroy($id);
    }

    /**
     * @param $id
     * @param array $columns
     * @return mixed
     */
    public function find($id, $columns = array('*')) {
        return $this->model->find($id, $columns);
    }

    /**
     * @param $attribute
     * @param $value
     * @param array $columns
     * @return mixed
     */
    public function findBy($attribute, $value, $columns = array('*')) {
        return $this->model->where($attribute, '=', $value)->first($columns);
    }

    /**
     * @return \Illuminate\Database\Eloquent\Builder
     * @throws RepositoryException
     */
    public function makeModel() {
        $model = $this->app->make($this->model());

        if (!$model instanceof Model)
            throw new RepositoryException("Class {$this->model()} must be an instance of Illuminate\\

        return $this->model = $model->newQuery();
    }
}
```

Pretty easy, right? Only thing left now is to inject *ActorRepository* in the *ActorsController*, or our business side of application:

```
<?php namespace App\Http\Controllers;

use App\Repositories\ActorRepository as Actor;

class ActorsController extends Controller {

    /**
     * @var Actor
     */
    private $actor;

    public function __construct(Actor $actor) {

        $this->actor = $actor;
    }

    public function index() {
        return \Response::json($this->actor->all());
    }
}
```

## Criteria Queries

As you can imagine, these basic actions are just enough for simple querying. For larger applications you'll most definitely need to make some custom queries to fetch more specific data set defined by some criteria.

To achieve this, we begin with defining what child (clients) criteria must implement. In other words, we'll create an abstract non instantiable class with just one method in it:

```
<?php namespace Bosnadev\Repositories\Criteria;

use Bosnadev\Repositories\Contracts\RepositoryInterface as Repository;
use Bosnadev\Repositories\Contracts\RepositoryInterface;

abstract class Criteria {

    /**
     * @param $model
     * @param RepositoryInterface $repository
     * @return mixed
     */
    public abstract function apply($model, Repository $repository);
}
```

This method will hold criteria query which will be applied in the *Repository* class on the concrete entity. We also need to extend our *Repository* class a bit to cover criteria queries. But first, let's create a new contract for the *Repository* class:

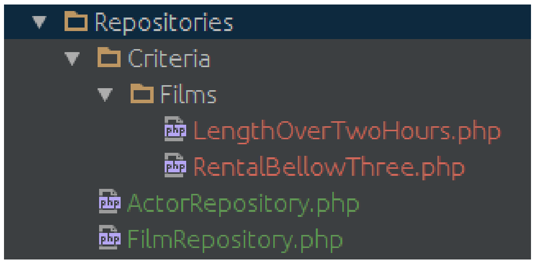
```
<?php namespace Bosnadev\Repositories\Contracts;

use Bosnadev\Repositories\Criteria\Criteria;

/**
 * Interface CriteriaInterface
 * @package Bosnadev\Repositories\Contracts
 */
interface CriteriaInterface {

    /**
     * @param bool $status
     * @return $this
     */
    public function skipCriteria($status = true);

    /**
     * @return mixed
     */
    public function getCriteria();

    /**
     * @param Criteria $criteria
     * @return $this
     */
    public function getByCriteria(Criteria $criteria);

    /**
     * @param Criteria $criteria
     * @return $this
     */
    public function pushCriteria(Criteria $criteria);

    /**
     * @return $this
     */
    public function  applyCriteria();
}
```

Now we can extend functionality of our *Repository* class by implementing *CriteriaInterface* contract:

```
<?php namespace Bosnadev\Repositories\Eloquent;

use Bosnadev\Repositories\Contracts\CriteriaInterface;
use Bosnadev\Repositories\Criteria\Criteria;
use Bosnadev\Repositories\Contracts\RepositoryInterface;
use Bosnadev\Repositories\Exceptions\RepositoryException;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Collection;
use Illuminate\Container\Container as App;

/**
 * Class Repository
 * @package Bosnadev\Repositories\Eloquent
 */
abstract class Repository implements RepositoryInterface, CriteriaInterface {

    /**
     * @var App
     */
    private $app;

    /**
     * @var
     */
    protected $model;

    /**
     * @var Collection
     */
    protected $criteria;

    /**
     * @var bool
     */
    protected $skipCriteria = false;

    /**
     * @param App $app
     * @param Collection $collection
     * @throws \Bosnadev\Repositories\Exceptions\RepositoryException
     */
    public function __construct(App $app, Collection $collection) {
        $this->app = $app;
        $this->criteria = $collection;
        $this->resetScope();
        $this->makeModel();
    }

    /**
     * Specify Model class name
     *
     * @return mixed
     */
    public abstract function model();

    /**
     * @param array $columns
     * @return mixed
     */
    public function all($columns = array('*')) {
        $this->applyCriteria();
        return $this->model->get($columns);
    }

    /**
     * @param int $perPage
     * @param array $columns
     * @return mixed
     */
    public function paginate($perPage = 1, $columns = array('*')) {
        $this->applyCriteria();
        return $this->model->paginate($perPage, $columns);
    }

    /**
     * @param array $data
     * @return mixed
     */
    public function create(array $data) {
        return $this->model->create($data);
    }

    /**
     * @param array $data
     * @param $id
     * @param string $attribute
     * @return mixed
     */
    public function update(array $data, $id, $attribute="id") {
        return $this->model->where($attribute, '=', $id)->update($data);
    }

    /**
     * @param $id
     * @return mixed
     */
    public function delete($id) {
        return $this->model->destroy($id);
    }

    /**
     * @param $id
     * @param array $columns
     * @return mixed
     */
    public function find($id, $columns = array('*')) {
        $this->applyCriteria();
        return $this->model->find($id, $columns);
    }

    /**
     * @param $attribute
     * @param $value
     * @param array $columns
     * @return mixed
     */
    public function findBy($attribute, $value, $columns = array('*')) {
        $this->applyCriteria();
        return $this->model->where($attribute, '=', $value)->first($columns);
    }

    /**
     * @return \Illuminate\Database\Eloquent\Builder
     * @throws RepositoryException
     */
    public function makeModel() {
        $model = $this->app->make($this->model());

        if (!$model instanceof Model)
            throw new RepositoryException("Class {$this->model()} must be an instance of Illuminate\\

        return $this->model = $model->newQuery();
    }

    /**
     * @return $this
     */
    public function resetScope() {
        $this->skipCriteria(false);
        return $this;
    }

    /**
     * @param bool $status
     * @return $this
     */
    public function skipCriteria($status = true){
        $this->skipCriteria = $status;
        return $this;
    }

    /**
     * @return mixed
     */
    public function getCriteria() {
        return $this->criteria;
    }

    /**
     * @param Criteria $criteria
     * @return $this
     */
    public function getByCriteria(Criteria $criteria) {
        $this->model = $criteria->apply($this->model, $this);
        return $this;
    }

    /**
     * @param Criteria $criteria
     * @return $this
     */
    public function pushCriteria(Criteria $criteria) {
        $this->criteria->push($criteria);
        return $this;
    }

    /**
     * @return $this
     */
    public function  applyCriteria() {
        if($this->skipCriteria === true)
            return $this;

        foreach($this->getCriteria() as $criteria) {
            if($criteria instanceof Criteria)
                $this->model = $criteria->apply($this->model, $this);
        }

        return $this;
    }
}
```

## Creating A New Criteria

With criteria queries, you can now organise your repositories more easily. Your repositories do not need to be thousands of lines long.



```
▼ Repositories
  ▼ Criteria
    ▼ Films
        LengthOverTwoHours.php
        RentalBellowThree.php
    ActorRepository.php
    FilmRepository.php
```

Your criteria class can look like this:

```php
<?php namespace App\Repositories\Criteria\Films;

use Bosnadev\Repositories\Contracts\CriteriaInterface;
use Bosnadev\Repositories\Contracts\RepositoryInterface as Repository;
use Bosnadev\Repositories\Contracts\RepositoryInterface;

class LengthOverTwoHours implements CriteriaInterface {

    /**
     * @param $model
     * @param RepositoryInterface $repository
     * @return mixed
     */
    public function apply($model, Repository $repository)
    {
        $query = $model->where('length', '>', 120);
        return $query;
    }
}
```

## Using Criteria In The Controller

Now when we have simple criteria, let’s see how we can use it. There is a two ways how you can apply the criteria on the repository. First is by using `pushCriteria()` method:

```php
<?php namespace App\Http\Controllers;

use App\Repositories\Criteria\Films\LengthOverTwoHours;
use App\Repositories\FilmRepository as Film;

class FilmsController extends Controller {

    /**
     * @var Film
     */
    private $film;

    public function __construct(Film $film) {

        $this->film = $film;
    }

    public function index() {
        $this->film->pushCriteria(new LengthOverTwoHours());
        return \Response::json($this->film->all());
    }
}
```

This method is useful if you need to apply multiple criteria, you can stack them as you wish. However, if you need to apply just one criteria, you can use `getByCriteria()` method:

```php
<?php namespace App\Http\Controllers;
```

```

use App\Repositories\Criteria\Films\LengthOverTwoHours;
use App\Repositories\FilmRepository as Film;

class FilmsController extends Controller {

    /**
     * @var Film
     */
    private $film;

    public function __construct(Film $film) {

        $this->film = $film;
    }

    public function index() {
        $criteria = new LengthOverTwoHours();
        return \Response::json($this->film->getByCriteria($criteria)->all());
    }
}
```

## Package Installation

You can install this package by adding this dependency in your composer require section:

```
"bosnadev/repositories": "0.*"
```

and just run `composer update` afterwards.

## Conclusion

Using repositories in your application have multiple benefits. From basic things like reducing code duplication and preventing you to make programming errors to making you application easier to extend, test and maintain.

From architectural point of view you managed to separate concerns. Your controller doesn't need to know how and where you store the data. Simple and beautiful. Abstract.

You can find this package on Github, where you can check for latest updates and bug fixes. I also plan to add new features like eager loading, caching and some configs so stay tuned by staring the repository. However, if you want to contribute in the development just fork the repository and send PR.

If you have any thoughts or suggestions, please let me know in the comment section bellow. See ya.

## Credits

This package is largely inspired by this great package by @andersao. Here is another package I used as reference. Also, I find these articles very helpful:

Creating flexible Controllers in Laravel 4 using Repositories
Laravel Repository Pattern
The Repository Pattern in Action
Laravel – Using Repository Pattern

About Latest Posts

**Mirza Pasic**
Full Stack Developer at OLX
Web Developer. Geek. Systematic. Dreamer

Follow me

**SHARE THIS:**

Facebook 219 | Google | Twitter | Reddit | Pocket | Email | Print

**LIKE THIS:**

Like

4 bloggers like this.

**RELATED**

**Using Repository Pattern In Laravel 5 - Eloquent Relations And Eager Loading**
March 26, 2015
In "Laravel"

**Deploy your Laravel 4 application on Heroku**
September 16, 2014
In "Laravel"

**Setting Up Laravel Environments**
December 26, 2014
In "Laravel"

March 7, 2015 Mirza Pasic Laravel design patterns, laravel

82 Comments Bosnadev Blog Login

Recommend 16 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

**Eden Chen** • 3 years ago

Hi, I'm now trying to use the repository mode in laravel, but I'm wondering what if I use multiple models in one repository, for instance, by default the PostRepository would only be related to the Post model, but if I would try to fetch a certain user's posts in the PostRepository, clearly I would need to use the User model to make the query condition, so should I just add a function like getUserPosts() in the PostRepository class and inject the User model to achieve my purpose(**fetch a certain user's posts**) or there're other better ways?

20 • Reply • Share ›

**ghostme** Eden Chen • 2 years ago

I think you can communicate between repositories. So basically after defining both the PostRepository and UserRepository class, you can call the getUserPost() method of the UserRepository class from the PostRepository

• Reply • Share ›

**Guido Contreras Woda** • 3 years ago

TL;DR:

I'd strongly suggest not using RepositoryInterface nor an abstract Repository class.

Here's why...

Let's talk abstraction through classification first. When you declare a RepositoryInterface, you are saying that all children of that interface (either abstract or concrete) will "be a Repository". But Repository is a pattern, not a classification. So being a pattern has no real meaning but the motivation behind the pattern itself, that will be communicated through the name or namespace anyway. In other words, "being a Repository" could be conveyed by naming a class "FilmRepository" without the need for a classification, as being a Repository is fullfilling a certain pattern's motivation, not being part of a certain group of things.

Now, let's talk about abstraction through generalization. The way your blog post starts, it's clear that you don't have a real motivation of generalizing: you start off with a general idea, an interface, but you don't really have use cases where this general idea came from. You assume that all children of this interface will need to provide at least those

see more

21 • Reply • Share ›

**chown** Guido Contreras Woda • 3 years ago

it would be good if you can show us some proper way then like how writer described his point of view in details above?

4 • Reply • Share ›

**Davide Pugliese** Guido Contreras Woda • 2 years ago

I was looking for info on this pattern and symfony and I read this post.
I partially agree with you about simplicity, the idea here however, is more to put together blocks of code that have a common topic. So we are gonna have with files FrontControllers in my case that just redirect calls from the view to the Repository and then to the Model (Entity). Now, I get your point that in some cases this might not be needed and we add one more "layer of complexity" sort to speak. However, think of an application as big as Facebook like the one I am working on right now, and I can guarantee you that having the code as well organized as possible is a desirable thing. The code ends up being more

reusable. You end up with more cohesive blocks of code.

1 ^ | ˅ • Reply • Share ›

**Guido Contreras Woda** ➦ Davide Pugliese • 2 years ago

Hi, Davide!
This is a two year old response, so I'll try and remember what I was thinking back then.

I would never argue against using the Repository pattern, it's a proven practice that helps projects organize their data access while also representing something in your domain. If used correctly, a repository can actually simplify interaction between domain services and entities.

What I argued back then, and I still believe in, is that you shouldn't take these kind of generalizations and bake them into any project. You should know about them and you should be watching over your code so you can abstract if and when you need to.

Most frameworks provide these abstractions anyway, so maybe I'm just being picky. But in my experience, I'd rather postpone writing all of this code until I need it. The best decision you can make depends on the amount of information you have, and being early in the project, you know nothing. Don't assume you'll need it.

1 ^ | ˅ • Reply • Share ›

**Albert Cloete** ➦ Guido Contreras Woda • 2 years ago

The way this article explains it is very similar to how Symfony implements it out of the box for you.

I agree with not writing things until you need them, but some basic things are very unlikely not be be used, like findById(), findAll(), etc. And even if you don't initially put them in an interface, I'd suggest putting it in the interface as soon as you do create those methods, just to make sure all your repositories stay uniform, and act predictably.

My feeling is that repositories become a mess without contracts.

^ | ˅ • Reply • Share ›

**rocketshipinspace** ➦ Guido Contreras Woda • 3 years ago

Absolutely agree with you here. I am terrible at putting my thoughts to words, but you've just described my personal opinion/findings perfectly.

^ | ˅ • Reply • Share ›

**Cătălin Georgescu** ➦ Guido Contreras Woda • 3 years ago

Any sample code to drive your point home? You made me curious.

^ | ˅ • Reply • Share ›

**Guido Contreras Woda** ➦ Cătălin Georgescu • 3 years ago

Hey! I'm back from vacations and swamped with work. I'll make a gist when I get some free time!

In the meantime, what are you interested in exactly? My point was that reusability through inheritance and sharing types between Repositories wasn't necessary, but if I were to make some sample code (like suggested maybe with the CrudQueryBuilder), it would look similar to the abstract Repository class implemented here. It's not that much about the code itself, but about how and when you choose your abstractions, and how you reuse your code.

Cheers!

2 • Reply • Share ›

**Bill Garrison** Guido Contreras Woda • 3 years ago

I would also be extremely interested in seeing an example of this. I'm still wrapping my head around repository pattern and examples help me a LOT

2 • Reply • Share ›

**Cătălin Georgescu** Guido Contreras Woda • 3 years ago

I'm fairly familiar with the pattern implemented here, but I would be interested in a more flexible way to do this, because being constrained to always implement the methods when you create a new repository for a new entity is not very cool, though Mirza's implementation helps a lot.
So when you have some time an example would be welcome.
And thanks for answering.

2 • Reply • Share ›

**Naren Chitrakar** Guido Contreras Woda • 3 years ago

So what you are saying basically is that the inheritance should never be used to reuse stuff and be solely used to represent something totally non-generic..in the case above..not to query database. And it seems like you are suggesting to let eloquent classes do what they do best in the concrete class in itself. But does not that makes the whole point of replacable data access layer a moot point?

• Reply • Share ›

**Guido Contreras Woda** Naren Chitrakar • 3 years ago

That's a great question!
It always depends on context. If we use inheritance to reuse code, we are coupling very hard to that code: we statically couple to a concrete class by inheriting, and we dynamically adhere to represent its parent because Liskov. How awkward could it be to receive an instance of `ActorRepository` when you were expecting a `FilmRepository`?
Of course this gets solved by type hinting the concrete child where you need it, which makes evident my initial point: there is no real use for the shared type.

Now, when I said Eloquent does this best, what I meant is that you don't need the common ground, because if you look at the proposed abstract `Repository`, all methods act as a proxy to the Eloquent instance. This is what I also meant when I analyzed abstraction through generalization: this is not a generalization at all. There is no real use case, you are just building Repositories imagining what you'll probably need from them. And, guess the acronym? Right! YAGNI.

If you aim to go the full RAD, fast development lane with no care for architecture and testing, then you're better off just using Eloquent through Facades, it will get you there faster.
If you aim to use the Repository pattern to abstract your data

access layer, my point is: you don't need common ground between your repositories. Implement data access methods only when you need them, and avoid overuse of inheritance when you actually want to decouple.

And, if you want to go a step beyond, check out www.laraveldoctrine.org and hit me at our Slack channel, maybe I get you interested in trying something different than Eloquent. ;-)

1 • Reply • Share ›

**Will** Guido Contreras Woda • 3 years ago

Just replying here so I get notified if/when you post an example, thanks :)

• Reply • Share ›

**Chu Quang Tu** Will • 2 years ago

One year ago and still no example =))) Talking is always the easiest part

6 • Reply • Share ›

**Aldrin Marquez** Chu Quang Tu • 2 months ago

ikr lmao

• Reply • Share ›

**Nguyễn Hiệp** Chu Quang Tu • 9 months ago

strongly agree =))

• Reply • Share ›

**ghostme** Chu Quang Tu • 2 years ago

lol

• Reply • Share ›

**developerbmw** • 3 years ago

"In other words, we allow business logic to access the data object without having knowledge of underlying data access architecture."

Yet your Repositories return Eloquent objects.

9 • Reply • Share ›

**Mateus Gomes** developerbmw • 3 years ago

There is a difference between business logic and application logic. Could not find any business login in the above examples.

And, the Eloquent model is returned because it is injected in the repository class. It could return anything we want.

1 • Reply • Share ›

**Nguyễn Hiệp** Mateus Gomes • 9 months ago

so if using Repository DP, we need to handle business logic completely beyond the application layer (in controller ?)

• Reply • Share ›

**developerbmw** Mateus Gomes • 3 years ago

If your repositories return Eloquent objects then code that uses the repository can easily become reliant on Eloquent. For example (taken from the code above):

```
public function index() {
$criteria = new LengthOverTwoHours();
return \Response::json($this->film->getByCriteria($criteria)->all());
}
```

It calls the all() method which is Eloquent-specific. What happens if you decide to change your repository to, for example, one that does raw SQL queries and returns POPOs or arrays? Suddenly your calls to all() won't work.

• Reply • Share ›

**Mateus Gomes** ➜ developerbmw • 3 years ago

The repository should return anything that is injected to it. If I am using Eloquent ORM, I will inject an eloquent model on it and use it within my repository.

In your example, in fact the all() method is an known method from eloquent, but I cant easily implement it on my repository so I can do this:

$this->filmRepository->getByCriteria($criteria)->all();

See the example:

```
class FilmRepository {
public function __construct(Film $film)
{
$this->film = $film;
}

public function getByCriteria($criteria)
```

**see more**

• Reply • Share ›

**Bishal Paudel** • a year ago

Do you actually require Criteria Interface and its implementations?
I would go by creating a bunch of criterias which are simple functions. I may need to group them by class, which is still unnecessary, especially when PHP does not require us to be pure OO, and provides plenty of room for functional programming.

3 • Reply • Share ›

**Ivana Momcilovic** • a year ago

Hi,
I have a specific situation where I need to execute one Criteria over the Repository and after that I need to get some results and call different SearchCriteria over the same Repo. Is there any chance to add something like removeCriteria as oposite of pushCriteria?

2 • Reply • Share ›

**isaackearl** • 3 years ago

Hi. I'm working on an API and realized recently that we need an abstraction layer for complex queries so we don't have to use the models directly in our controller. Your repository package has caught my eye and I've started to work on implementing it.

I'm wondering if there is a way with criteria to pass in parameters? I want to create criteria but I want them to be a bit more flexible, and I don't understand from the examples if this is possible.

For example I'm currently using eloquent scope queries to handle custom criteria. I want to move that logic to my repository through criteria... and I'm not sure if it is possible.

```
public function scopeBeforeId($query, $id)
{
if ($id > 0) {
return $query->where('id', '<', $id);
}
return $query;
}
```

in the example I see a way to make a similiar criteria with a hardcoded $id... is it possible to pass variables to the criteria when you apply them?

second quesiton: Is it possible to load certain criteria on every load of a particular repository? To do that do I just override the constructor and add criteria there?

thanks! and great work.

2 ^ | ˅ • Reply • Share ›

**tooleks** ➦ isaackearl • 2 years ago

Yes, you can pass parameters into the criteria constructor method.

```
// Defining the criteria.
class LengthOverHours implements CriteriaInterface {
private $hours;

public function __construct($hours) {
$this->hours = $hours;
}

public function apply($model, Repository $repository)
{
$query = $model->where('length', '>', $this->hours);
return $query;
}
}

// Passing variable into the criteria.
$this->filmRepository->pushCriteria(new LengthOverHours(2));
$films = $this->filmRepository->all();
```

1 ^ | ˅ • Reply • Share ›

**Ngoc Phan** • 3 years ago

Perfect article

a little bug

LengthOverTwoHours implements CriteriaInterface

should be

LengthOverTwoHours extends Criteria

1 ^ | ˅ • Reply • Share ›

**Teej Ten** ➦ Ngoc Phan • 2 years ago

Thank you soooooooo much Ngoc! I couldn't get an error when I used this code just a bunch of html output gibberish so it was super hard to debug. It is so obvious now why things were not working! Thanks once again, best regards!

Tim

^ | v • Reply • Share ›

**Tim Sims** • 3 years ago

Hi, how can you cache result (I'm adding a Cache layer on Repository ) with criteria queues?

Without criteria, I can easily generate a cache key by the params, and cache data in it. So next time I pass the same params I can fetch cache from the same key

But with criteria, I can't access to those addiction params

1 ^ | v • Reply • Share ›

**Mirza Pasic** Mod → Tim Sims • 3 years ago

Hi Tim, can you provide me an example for that?

^ | v • Reply • Share ›

**Tim Sims** → Mirza Pasic • 3 years ago

Hi, not sure how to write code in Disqus, so I use gist for better explanation

https://gist.github.com/tim...

^ | v • Reply • Share ›

**Spir** → Tim Sims • 3 years ago

I'm doing almost the same thing. But I use the decorator pattern to decorate the repository with an added cache feature. Check it out: http://laravel.io/bin/Nkb4e

^ | v • Reply • Share ›

**Alen Abdula** • 3 years ago

Mirza svaka cast!!!

1 ^ | v • Reply • Share ›

**Anderson Andrade** • 4 years ago

Great article! I think you drew in my package http://bit.ly/1EuZWyu ;) . I liked the changes you made .

1 ^ | v • Reply • Share ›

**Mirza Pasic** Mod → Anderson Andrade • 4 years ago

Actually, your package has been the inspiration for something I have not added yet (not completly at least). Request criteria queries :) You gave me an idea how to improve that class. I'll be working on that in the next few days ;)

This was initial inspiration for criteria queries https://github.com/anlutro/...

But, I think you did the better job with criteria... I'll update the readme to credit you both for that influence ;)

2 ^ | v • Reply • Share ›

**Anderson Andrade** → Mirza Pasic • 4 years ago

Thanks! Will contribute to your project. ;)

^ | v • Reply • Share ›

**thbt** • 3 years ago

pattern isn't something everyone should use in all their projects without thinking whether it's beneficial for their particular situation. There is a tendency for some devs to try to apply every pattern they learn to every situation.

So keep in mind that using a repository adds a significant amount of complexity above simply using something like Eloquent directly, especially if you try to completely isolate the model so that all interactions have to be done through the repository. There are situations where you need that separating barrier and you really want a layer of isolation between your business logic and the data source. And the Repository is something that web devs should at least be aware of so they know to use it when they need it. But if your app doesn't need that separation, adding it for no reason can make a simple app far more complicated than it needs to be.

1 • Reply • Share ›

**Mateus Gomes** thbt • 3 years ago

I do agree that using Eloquent directly is far more simple. But it gets really bad when you decide to change from Eloquent to another ORM or don't use an ORM at all. Please keep that in mind.
Personally I prefer make things a little bit harder first so I won't (possibly) waste a lot of time later.

• Reply • Share ›

**Jay Official** thbt • 3 years ago

Very True! couldn't agree more

• Reply • Share ›

**Connor Leech** • 2 months ago

Do you have any more resources available to understand how eager loading works within the repository pattern?

• Reply • Share ›

**Rob Bennett** • 5 months ago

Hi, just came across this, great pattern, thanks!

Just a quick question, shouldn't the resetScope method of the repository set the criteria collection to an empty collection and call $this->makeModel() again? Doing this allows for the query scope to be reset on the model instance within a session.

• Reply • Share ›

**Firman Taruna Nugraha** • 7 months ago

your presentation is good man, thanks

• Reply • Share ›

**Francis Rodrigues** • a year ago

Could you please help me develop a relationship search for another model (related table)?
I think we could do more sophisticated searches using "with ()" method, etc.

• Reply • Share ›

**Francis Rodrigues** • a year ago