




 andersao / **IS-repository**


 Watch133


 Star2,134


 Fork454

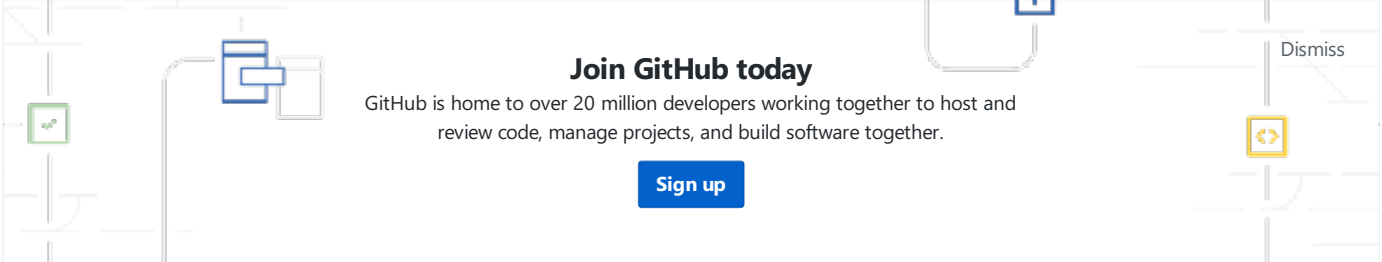
 Code

 Issues101

 Pull requests20

 Projects0

 Insights



Join GitHub today

GitHub is home to over 20 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)

Dismiss

Laravel 5 - Repositories to abstract the database layer

📄 412 commits

🌿 6 branches

📦 67 releases


👤 69 contributors

Branch: master

New pull request


Find file

Clone or download

 eullercdr Merge pull request #487 from nabeelio/fix-paginate-declaration

Latest commit 201039a Dec 14, 2017

src	Merge pull request #487 from nabeelio/fix-paginate-declaration	Dec 14, 2017
.editorconfig	Update composer.json for support Laravel 5.2	Dec 24, 2015
.gitignore	Update composer.json for support Laravel 5.2	Dec 24, 2015
CONTRIBUTING.md	added contribution guide	Apr 1, 2016
LICENSE.md	added license	Apr 1, 2016
README.md	update app()->path to app()->path().	Nov 13, 2017
composer.json	add provider	Oct 11, 2017
composer.lock	Adding validator package as required since we have the validator gene...	Apr 1, 2016
migration-to-2.0.md	Update README	Apr 4, 2015
migration-to-2.1.md	Prepare to release 2.1	Jun 27, 2015
phpunit.xml	Publishe papackage	Feb 12, 2015

 README.md

Laravel 5 Repositories

Laravel 5 Repositories is used to abstract the data layer, making our application more flexible to maintain.

stable2.6.29

downloads612.56 k

unstabledev-master

licenseMIT

analyticsGA

maintainabilityC

See versions: **1.0.* / 2.0.***

Migrate to: **2.0 / 2.1**

You want to know a little more about the Repository pattern? [Read this great article.](#)

Table of Contents

- Installation
 - Composer
 - Laravel

- [Methods](#)
 - [RepositoryInterface](#)
 - [RepositoryCriteriaInterface](#)
 - [CacheableInterface](#)
 - [PresenterInterface](#)
 - [CriteriaInterface](#)
- [Usage](#)
 - [Create a Model](#)
 - [Create a Repository](#)
 - [Generators](#)
 - [Use methods](#)
 - [Create a Criteria](#)
 - [Using the Criteria in a Controller](#)
 - [Using the RequestCriteria](#)
- [Cache](#)
 - [Usage](#)
 - [Config](#)
- [Validators](#)
 - [Using a Validator Class](#)
 - [Create a Validator](#)
 - [Enabling Validator in your Repository](#)
 - [Defining rules in the repository](#)
- [Presenters](#)
 - [Fractal Presenter](#)
 - [Create a Fractal Presenter](#)
 - [Model Transformable](#)
 - [Enabling in your Repository](#)

Installation

Composer

Execute the following command to get the latest version of the package:

```
composer require prettus/l5-repository
```

Laravel

>= laravel5.5

ServiceProvider will be attached automatically

Other

In your `config/app.php` add `Prettus\Repository\Providers\RepositoryServiceProvider::class` to the end of the `providers` array:

```
'providers' => [
    ...
    Prettus\Repository\Providers\RepositoryServiceProvider::class,
],
```

If Lumen

```
$app->register(Prettus\Repository\Providers\LumenRepositoryServiceProvider::class);
```

Publish Configuration

```
php artisan vendor:publish --provider "Prettus\Repository\Providers\RepositoryServiceProvider"
```

Methods

Prettus\Repository\Contracts\RepositoryInterface

- all(\$columns = array('*'))
- first(\$columns = array('*'))
- paginate(\$limit = null, \$columns = ['*'])
- find(\$id, \$columns = ['*'])
- findByField(\$field, \$value, \$columns = ['*'])
- findWhere(array \$where, \$columns = ['*'])
- findWhereIn(\$field, array \$where, \$columns = ['*'])
- findWhereNotIn(\$field, array \$where, \$columns = ['*'])
- create(array \$attributes)
- update(array \$attributes, \$id)
- updateOrCreate(array \$attributes, array \$values = [])
- delete(\$id)
- deleteWhere(array \$where)
- orderBy(\$column, \$direction = 'asc');
- with(array \$relations);
- has(string \$relation);
- whereHas(string \$relation, closure \$closure);
- hidden(array \$fields);
- visible(array \$fields);
- scopeQuery(Closure \$scope);
- getFieldsSearchable();
- setPresenter(\$presenter);
- skipPresenter(\$status = true);

Prettus\Repository\Contracts\RepositoryCriteriaInterface

- pushCriteria(\$criteria)
- popCriteria(\$criteria)
- getCriteria()
- getByCriteria(CriteriaInterface \$criteria)
- skipCriteria(\$status = true)
- getFieldsSearchable()

Prettus\Repository\Contracts\CacheableInterface

- setCacheRepository(CacheRepository \$repository)
- getCacheRepository()
- getCacheKey(\$method, \$args = null)
- getCacheMinutes()
- skipCache(\$status = true)

Prettus\Repository\Contracts\PresenterInterface

- present(\$data);

Prettus\Repository\Contracts\Presentable

- setPresenter(PresenterInterface \$presenter);
- presenter();

Prettus\Repository\Contracts\CriterialInterface

- apply(\$model, RepositoryInterface \$repository);

Prettus\Repository\Contracts\Transformable

- transform();

Usage

Create a Model

Create your model normally, but it is important to define the attributes that can be filled from the input form data.

```
namespace App;

class Post extends Eloquent { // or Ardent, Or any other Model Class

    protected $fillable = [
        'title',
        'author',
        ...
    ];

    ...
}
```

Create a Repository

```
namespace App;

use Prettus\Repository\Eloquent\BaseRepository;

class PostRepository extends BaseRepository {

    /**
     * Specify Model class name
     *
     * @return string
     */
    function model()
    {
        return "App\Post";
    }
}
```

Generators

Create your repositories easily through the generator.

Config

You must first configure the storage location of the repository files. By default is the "app" folder and the namespace "App".

Please note that, values in the `paths` array are acutally used as both *namespace* and file paths. Relax though, both foreward and backward slashes are taken care of during generation.

```
...
'generator'=>[
  'basePath'=>app()->path(),
  'rootNamespace'=>'App\\',
  'paths'=>[
    'models' => 'Entities',
    'repositories' => 'Repositories',
    'interfaces' => 'Repositories',
    'transformers' => 'Transformers',
    'presenters' => 'Presenters',
    'validators' => 'Validators',
    'controllers' => 'Http/Controllers',
    'provider' => 'RepositoryServiceProvider',
    'criteria' => 'Criteria',
  ]
]
```

You may want to save the root of your project folder out of the app and add another namespace, for example

```
...
'generator'=>[
  'basePath' => base_path('src/Lorem'),
  'rootNamespace' => 'Lorem\\'
]
```

Additionally, you may wish to customize where your generated classes end up being saved. That can be accomplished by editing the `paths` node to your liking. For example:

```
'generator'=>[
  'basePath'=>app()->path(),
  'rootNamespace'=>'App\\',
  'paths'=>[
    'models'=>'Models',
    'repositories'=>'Repositories\\Eloquent',
    'interfaces'=>'Contracts\\Repositories',
    'transformers'=>'Transformers',
    'presenters'=>'Presenters',
    'validators' => 'Validators',
    'controllers' => 'Http/Controllers',
    'provider' => 'RepositoryServiceProvider',
    'criteria' => 'Criteria',
  ]
]
```

Commands

To generate everything you need for your Model, run this command:

```
php artisan make:entity Post
```

This will create the Controller, the Validator, the Model, the Repository, the Presenter and the Transformer classes. It will also create a new service provider that will be used to bind the Eloquent Repository with its corresponding Repository Interface. To load it, just add this to your `AppServiceProvider@register` method:

```
$this->app->register(RepositoryServiceProvider::class);
```

You can also pass the options from the `repository` command, since this command is just a wrapper.

To generate a repository for your Post model, use the following command

```
php artisan make:repository Post
```

To generate a repository for your Post model with Blog namespace, use the following command

```
php artisan make:repository "Blog\Post"
```

Added fields that are fillable

```
php artisan make:repository "Blog\Post" --fillable="title,content"
```

To add validations rules directly with your command you need to pass the `--rules` option and create migrations as well:

```
php artisan make:entity Cat --fillable="title:string,content:text" --rules="title=>required|min:2, content=>s
```

The command will also create your basic RESTfull controller so just add this line into your `routes.php` file and you will have a basic CRUD:

```
Route::resource('cats', CatsController::class);
```

When running the command, you will be creating the "Entities" folder and "Repositories" inside the folder that you set as the default.

Done, done that just now you do bind its interface for your real repository, for example in your own Repositories Service Provider.

```
App::bind('{YOUR_NAMESPACE}Repositories\PostRepository', '{YOUR_NAMESPACE}Repositories\PostRepositoryEloquent
```

And use

```
public function __construct({YOUR_NAMESPACE}Repositories\PostRepository $repository){  
    $this->repository = $repository;  
}
```

Alternatively, you could use the artisan command to do the binding for you.

```
php artisan make:bindings Cats
```

Use methods

```
namespace App\Http\Controllers;  
  
use App\PostRepository;  
  
class PostsController extends BaseController {  
  
    /**  
     * @var PostRepository  
     */  
    protected $repository;  
  
    public function __construct(PostRepository $repository){  
        $this->repository = $repository;  
    }  
  
    ....  
}
```

Find all results in Repository

```
$posts = $this->repository->all();
```

Find all results in Repository with pagination

```
$posts = $this->repository->paginate($limit = null, $columns = ['*']);
```

Find by result by id

```
$post = $this->repository->find($id);
```

Hiding attributes of the model

```
$post = $this->repository->hidden(['country_id'])->find($id);
```

Showing only specific attributes of the model

```
$post = $this->repository->visible(['id', 'state_id'])->find($id);
```

Loading the Model relationships

```
$post = $this->repository->with(['state'])->find($id);
```

Find by result by field name

```
$posts = $this->repository->findByField('country_id', '15');
```

Find by result by multiple fields

```
$posts = $this->repository->findWhere([
    //Default Condition =
    'state_id'=>'10',
    'country_id'=>'15',
    //Custom Condition
    ['columnName', '>', '10']
]);
```

Find by result by multiple values in one field

```
$posts = $this->repository->findWhereIn('id', [1,2,3,4,5]);
```

Find by result by excluding multiple values in one field

```
$posts = $this->repository->findWhereNotIn('id', [6,7,8,9,10]);
```

Find all using custom scope

```
$posts = $this->repository->scopeQuery(function($query){
    return $query->orderBy('sort_order', 'asc');
})->all();
```

Create new entry in Repository

```
$post = $this->repository->create( Input::all() );
```

Update entry in Repository

```
$post = $this->repository->update( Input::all(), $id );
```

Delete entry in Repository

```
$this->repository->delete($id)
```

Delete entry in Repository by multiple fields

```
$this->repository->deleteWhere([
    //Default Condition =
    'state_id'=>'10',
    'country_id'=>'15',
])
```

Create a Criteria

Using the command

```
php artisan make:criteria My
```

Criteria are a way to change the repository of the query by applying specific conditions according to your needs. You can add multiple Criteria in your repository.

```
use Prettus\Repository\Contracts\RepositoryInterface;
use Prettus\Repository\Contracts/CriteriaInterface;

class MyCriteria implements CriteriaInterface {

    public function apply($model, RepositoryInterface $repository)
    {
        $model = $model->where('user_id', '=', Auth::user()->id );
        return $model;
    }
}
```

Using the Criteria in a Controller

```
namespace App\Http\Controllers;

use App\PostRepository;

class PostsController extends BaseController {

    /**
     * @var PostRepository
     */
    protected $repository;

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

    public function index()
    {
        $this->repository->pushCriteria(new MyCriteria1());
        $this->repository->pushCriteria(MyCriteria2::class);
        $posts = $this->repository->all();
        ...
    }
}
```

Getting results from Criteria

```
$posts = $this->repository->getByCriteria(new MyCriteria());
```


Setting the default Criteria in Repository

```
use Prettus\Repository\Eloquent\BaseRepository;

class PostRepository extends BaseRepository {

    public function boot(){
        $this->pushCriteria(new MyCriteria());
        // or
        $this->pushCriteria(AnotherCriteria::class);
        ...
    }

    function model(){
        return "App\\Post";
    }
}
```

Skip criteria defined in the repository

Use `skipCriteria` before any other chaining method

```
$posts = $this->repository->skipCriteria()->all();
```

Popping criteria

Use `popCriteria` to remove a criteria

```
$this->repository->popCriteria(new Criterial1());
// or
$this->repository->popCriteria(Criterial1::class);
```

Using the RequestCriteria

RequestCriteria is a standard Criteria implementation. It enables filters to perform in the repository from parameters sent in the request.

You can perform a dynamic search, filter the data and customize the queries.

To use the Criteria in your repository, you can add a new criteria in the boot method of your repository, or directly use in your controller, in order to filter out only a few requests.

Enabling in your Repository

```

use Prettus\Repository\Eloquent\BaseRepository;
use Prettus\Repository\Criteria\RequestCriteria;

class PostRepository extends BaseRepository {

    /**
     * @var array
     */
    protected $fieldSearchable = [
        'name',
        'email'
    ];

    public function boot(){
        $this->pushCriteria(app('Prettus\Repository\Criteria\RequestCriteria'));
        ...
    }

    function model(){
        return "App\\Post";
    }
}

```

Remember, you need to define which fields from the model can be searchable.

In your repository set **\$fieldSearchable** with the name of the fields to be searchable or a relation to fields.

```

protected $fieldSearchable = [
    'name',
    'email',
    'product.name'
];

```

You can set the type of condition which will be used to perform the query, the default condition is "="

```

protected $fieldSearchable = [
    'name'=>'like',
    'email', // Default Condition "="
    'your_field'=>'condition'
];

```

Enabling in your Controller

```

public function index()
{
    $this->repository->pushCriteria(app('Prettus\Repository\Criteria\RequestCriteria'));
    $posts = $this->repository->all();
    ...
}

```

Example the Criteria

Request all data without filter by request

```
http://prettus.local/users
```

```
[
  {
    "id": 1,
    "name": "John Doe",
    "email": "john@gmail.com",
    "created_at": "-0001-11-30 00:00:00",
    "updated_at": "-0001-11-30 00:00:00"
  },
  {
    "id": 2,
    "name": "Lorem Ipsum",
    "email": "lorem@ipsum.com",
    "created_at": "-0001-11-30 00:00:00",
    "updated_at": "-0001-11-30 00:00:00"
  },
  {
    "id": 3,
    "name": "Laravel",
    "email": "laravel@gmail.com",
    "created_at": "-0001-11-30 00:00:00",
    "updated_at": "-0001-11-30 00:00:00"
  }
]
```

Conducting research in the repository

```
http://prettus.local/users?search=John%20Doe
```

or

```
http://prettus.local/users?search=John&searchFields=name:like
```

or

```
http://prettus.local/users?search=john@gmail.com&searchFields=email:=
```

or

```
http://prettus.local/users?search=name:John Doe;email:john@gmail.com
```

or

```
http://prettus.local/users?search=name:John;email:john@gmail.com&searchFields=name:like;email:=
```

```
[
  {
    "id": 1,
    "name": "John Doe",
    "email": "john@gmail.com",
    "created_at": "-0001-11-30 00:00:00",
    "updated_at": "-0001-11-30 00:00:00"
  }
]
```

By default RequestCriteria makes its queries using the **OR** comparison operator for each query parameter.

```
http://prettus.local/users?search=age:17;email:john@gmail.com
```

The above example will execute the following query:

```
SELECT * FROM users WHERE age = 17 OR email = 'john@gmail.com';
```

In order for it to query using the **AND**, pass the *searchJoin* parameter as shown below:

```
http://prettus.local/users?search=age:17;email:john@gmail.com&searchJoin=and
```

Filtering fields

```
http://prettus.local/users?filter=id;name
```

```
[
  {
    "id": 1,
    "name": "John Doe"
  },
  {
    "id": 2,
    "name": "Lorem Ipsum"
  },
  {
    "id": 3,
    "name": "Laravel"
  }
]
```

Sorting the results

```
http://prettus.local/users?filter=id;name&orderBy=id&sortedBy=desc
```

```
[
  {
    "id": 3,
    "name": "Laravel"
  },
  {
    "id": 2,
    "name": "Lorem Ipsum"
  },
  {
    "id": 1,
    "name": "John Doe"
  }
]
```

Sorting through related tables

```
http://prettus.local/users?orderBy=posts|title&sortedBy=desc
```

Query will have something like this

```
...
INNER JOIN posts ON users.post_id = posts.id
...
ORDER BY title
...
```

```
http://prettus.local/users?orderBy=posts:custom_id|posts.title&sortedBy=desc
```

Query will have something like this

```
...
INNER JOIN posts ON users.custom_id = posts.id
...
ORDER BY posts.title
...
```

Add relationship

```
http://prettus.local/users?with=groups
```

Overwrite params name

You can change the name of the parameters in the configuration file **config/repository.php**

Cache

Add a layer of cache easily to your repository

Cache Usage

Implements the interface CacheableInterface and use CacheableRepository Trait.

```
use Prettus\Repository\Eloquent\BaseRepository;
use Prettus\Repository\Contracts\CacheableInterface;
use Prettus\Repository\Traits\CacheableRepository;

class PostRepository extends BaseRepository implements CacheableInterface {

    use CacheableRepository;

    ...

}
```

Done , done that your repository will be cached , and the repository cache is cleared whenever an item is created, modified or deleted.

Cache Config

You can change the cache settings in the file *config/repository.php* and also directly on your repository.

config/repository.php

```
'cache'=>[
    //Enable or disable cache repositories
    'enabled' => true,

    //Lifetime of cache
    'minutes' => 30,

    //Repository Cache, implementation Illuminate\Contracts\Cache\Repository
    'repository'=> 'cache',

    //Sets clearing the cache
    'clean' => [
        //Enable, disable clearing the cache on changes
        'enabled' => true,

        'on' => [
            //Enable, disable clearing the cache when you create an item
            'create'=>true,

            //Enable, disable clearing the cache when upgrading an item
            'update'=>true,

            //Enable, disable clearing the cache when you delete an item
            'delete'=>true,
        ]
    ],
    'params' => [
        //Request parameter that will be used to bypass the cache repository
        'skipCache'=>'skipCache'
    ],
    'allowed'=>[
        //Allow caching only for some methods
        'only' =>null,

        //Allow caching for all available methods, except
        'except'=>null
    ],
],
```

It is possible to override these settings directly in the repository.

```

use Prettus\Repository\Eloquent\BaseRepository;
use Prettus\Repository\Contracts\CacheableInterface;
use Prettus\Repository\Traits\CacheableRepository;

class PostRepository extends BaseRepository implements CacheableInterface {

    // Setting the lifetime of the cache to a repository specifically
    protected $cacheMinutes = 90;

    protected $cacheOnly = ['all', ...];
    //or
    protected $cacheExcept = ['find', ...];

    use CacheableRepository;

    ...
}

```

The cacheable methods are : all, paginate, find, findByField, findWhere, getByCriteria

Validators

Requires [prettus/laravel-validator](#). `composer require prettus/laravel-validator`

Easy validation with `prettus/laravel-validator`

[For more details click here](#)

Using a Validator Class

Create a Validator

In the example below, we define some rules for both creation and edition

```

use \Prettus\Validator\LaravelValidator;

class PostValidator extends LaravelValidator {

    protected $rules = [
        'title' => 'required',
        'text'  => 'min:3',
        'author'=> 'required'
    ];

}

```

To define specific rules, proceed as shown below:

```

use \Prettus\Validator\Contracts\ValidatorInterface;
use \Prettus\Validator\LaravelValidator;

class PostValidator extends LaravelValidator {

    protected $rules = [
        ValidatorInterface::RULE_CREATE => [
            'title' => 'required',
            'text'  => 'min:3',
            'author'=> 'required'
        ],
        ValidatorInterface::RULE_UPDATE => [
            'title' => 'required'
        ]
    ];

}

```

Enabling Validator in your Repository

```

use Prettus\Repository\Eloquent\BaseRepository;
use Prettus\Repository\Criteria\RequestCriteria;

class PostRepository extends BaseRepository {

    /**
     * Specify Model class name
     *
     * @return mixed
     */
    function model(){
        return "App\\Post";
    }

    /**
     * Specify Validator class name
     *
     * @return mixed
     */
    public function validator()
    {
        return "App\\PostValidator";
    }
}

```

Defining rules in the repository

Alternatively, instead of using a class to define its validation rules, you can set your rules directly into the rules repository property, it will have the same effect as a Validation class.

```

use Prettus\Repository\Eloquent\BaseRepository;
use Prettus\Repository\Criteria\RequestCriteria;
use Prettus\Validator\Contracts\ValidatorInterface;

class PostRepository extends BaseRepository {

    /**
     * Specify Validator Rules
     * @var array
     */
    protected $rules = [
        ValidatorInterface::RULE_CREATE => [
            'title' => 'required',
            'text' => 'min:3',
            'author' => 'required'
        ],
        ValidatorInterface::RULE_UPDATE => [
            'title' => 'required'
        ]
    ];

    /**
     * Specify Model class name
     *
     * @return mixed
     */
    function model(){
        return "App\\Post";
    }
}

```

Validation is now ready. In case of a failure an exception will be given of the type:

Prettus\Validator\Exceptions\ValidatorException

Presenters

Presenters function as a wrapper and renderer for objects.

Fractal Presenter

Requires [Fractal](#). `composer require league/fractal`

There are two ways to implement the Presenter, the first is creating a TransformerAbstract and set it using your Presenter class as described in the Create a Transformer Class.

The second way is to make your model implement the Transformable interface, and use the default Presenter ModelFractalPresenter, this will have the same effect.

Transformer Class

Create a Transformer using the command

```
php artisan make:transformer Post
```

This will generate the class beneath.

Create a Transformer Class

```
use League\Fractal\TransformerAbstract;

class PostTransformer extends TransformerAbstract
{
    public function transform(\Post $post)
    {
        return [
            'id'      => (int) $post->id,
            'title'   => $post->title,
            'content' => $post->content
        ];
    }
}
```

Create a Presenter using the command

```
php artisan make:presenter Post
```

The command will prompt you for creating a Transformer too if you haven't already.

Create a Presenter

```
use Prettus\Repository\Presenter\FractalPresenter;

class PostPresenter extends FractalPresenter {

    /**
     * Prepare data to present
     *
     * @return \League\Fractal\TransformerAbstract
     */
    public function getTransformer()
    {
        return new PostTransformer();
    }
}
```

Enabling in your Repository


```

use Prettus\Repository\Eloquent\BaseRepository;

class PostRepository extends BaseRepository {

    ...

    public function presenter()
    {
        return "App\\Presenter\\PostPresenter";
    }
}

```

Or enable it in your controller with

```

$this->repository->setPresenter("App\\Presenter\\PostPresenter");

```

Using the presenter after from the Model

If you recorded a presenter and sometime used the `skipPresenter()` method or simply you do not want your result is not changed automatically by the presenter. You can implement Presentable interface on your model so you will be able to present your model at any time. See below:

In your model, implement the interface `Prettus\Repository\Contracts\Presentable` and `Prettus\Repository\Traits\PresentableTrait`

```

namespace App;

use Prettus\Repository\Contracts\Presentable;
use Prettus\Repository\Traits\PresentableTrait;

class Post extends Eloquent implements Presentable {

    use PresentableTrait;

    protected $fillable = [
        'title',
        'author',
        ...
    ];

    ...

}

```

There, now you can submit your Model individually, See an example:

```

$repository = app('App\PostRepository');
$repository->setPresenter("Prettus\\Repository\\Presenter\\ModelFractalPresenter");

//Getting the result transformed by the presenter directly in the search
$post = $repository->find(1);

print_r( $post ); //It produces an output as array

...

//Skip presenter and bringing the original result of the Model
$post = $repository->skipPresenter()->find(1);

print_r( $post ); //It produces an output as a Model object
print_r( $post->presenter() ); //It produces an output as array

```

You can skip the presenter at every visit and use it on demand directly into the model, for it set the `$skipPresenter` attribute to true in your repository:

```

use Prettus\Repository\Eloquent\BaseRepository;

class PostRepository extends BaseRepository {

    /**
     * @var bool
     */
    protected $skipPresenter = true;

    public function presenter()
    {
        return "App\\Presenter\\PostPresenter";
    }
}

```

Model Class

Implement Interface

```

namespace App;

use Prettus\Repository\Contracts\Transformable;

class Post extends Eloquent implements Transformable {
    ...
    /**
     * @return array
     */
    public function transform()
    {
        return [
            'id'      => (int) $this->id,
            'title'   => $this->title,
            'content' => $this->content
        ];
    }
}

```

Enabling in your Repository

`Prettus\Repository\Presenter\ModelFractalPresenter` is a Presenter default for Models implementing Transformable

```

use Prettus\Repository\Eloquent\BaseRepository;

class PostRepository extends BaseRepository {

    ...

    public function presenter()
    {
        return "Prettus\\Repository\\Presenter\\ModelFractalPresenter";
    }
}

```

Or enable it in your controller with

```

$this->repository->setPresenter("Prettus\\Repository\\Presenter\\ModelFractalPresenter");

```

Skip Presenter defined in the repository

Use *skipPresenter* before any other chaining method

```

$posts = $this->repository->skipPresenter()->all();

```

or

```
$this->repository->skipPresenter();  
  
$posts = $this->repository->all();
```

