



PHP

Laravel – Using Repository Pattern

[Laravel](#) is one of the most popular PHP MVC frameworks and taking the [Php](#) community rapidly than any other frameworks probably couldn't do and it's because of a great combination of power, extensibility and easiness. The framework provides so many ways to a developer to develop an application using one or another, depending on the size of the project developer can use any possible way that Laravel provides and project of any size fits well in [Laravel](#). One of the most popular ways for building an application using [Laravel](#) is the [Repository Pattern](#) and use of this pattern has a lots of benefits and most of the developers follow this pattern to build an application using [Laravel](#).

The fundamental idea or the primary goal to use this pattern in a [Laravel](#) application is to create a bridge or link between application models and controllers. In other words, to decouple the hard dependencies of models from the controllers. In a typical Laravel application one may use some thing like this in a controller:

```
1 class UserController extends BaseController {
2
3     public function getAllUsers()
4     {
5         $users = User::all();
6         return View::make('user.index', compact('users'));
7     }
8
9 }
```

There is nothing new for a [Laravel](#) user, in that controller method, the [User](#) model is being directly used to get all the users from the database and this [User](#) model basically extends the Laravel's [Eloquent ORM](#) which is a common approach in any [Laravel](#) application because models that interact with database extends the [Eloquent](#).

So far, everything is good and this code is correct and will work as expected but still the code is not quite perfect and it's using the `User` model directly from the controller and manual access to the data layer from a controller violates the law of good design. It's bad and known as anti pattern because the controller has hard dependencies on the `User` model and hence it's tightly coupled to each other and that's why it's hard to test (unit testing). To, overcome this problem, a repository comes in to the play and a repository is just a normal class with methods to access the data layer (`User` model here) and the controller uses this repository to interact with the model instead of directly using the model by itself. This is an example of a repository (`UserRepository`) that the `UserController` can use to interact with the `User` model:

```
1 class UserRepository {
2
3     public function getAllUsers()
4     {
5         return User::all();
6     }
7
8 }
```

This repository could be injected into the `UserController` during the initiation of the class using the `__construct()` method and this is possible by type hinting the `UserRepository` class like this:

```
1 class UserController extends BaseController {
2
3     protected $user = null;
4
5     public function __construct(UserRepository $user)
6     {
7         $this->user = $user;
8     }
9
10    public function showUsers()
11    {
12        $users = $this->user->getAllUsers();
13        return View::make('user.index', compact('users'));
14    }
15
16 }
```

This is the `UserController` class now and it has a dependency on `UserRepository` and this dependency would be injected into the class during initialization by the smart `IoC` container component of the framework automatically. While this will work and hard coded dependency is replaced using the repository but still now the `UserController` class is indirectly dependent on the `User` model because the `UserRepository` class depends on that model and testing is still hard. So, to overcome this situation, an `Interface` can come to the play. While the primary goal of an interface is to build a contract with the class that implements the `interface` and it's a very commonly used approach, in most cases for public `API` to ensure the correctness of an application that uses the public `API` where an interface defines the behavior of a class by declaring methods and the class that implements the interface must implements all the methods declared in the interface.

An interface says, "This is what all classes that implement this particular interface will look like." Thus, any code that uses a particular interface knows what methods can be called for that interface, and that's all. So the interface is used to establish a protocol between classes.

While this is true that an interface is like a contract but also there are other things an interface does. It is being used as an abstraction layer between the consumer class (UserController that will use the interface) and the concrete class. Here, concrete class refers to the class that implements the interface (an abstraction of the concrete class). In other words an interface hides the details from the consumer class when the interface is used to type hint the data type as a dependency of the consumer class. Let's see an example to make it clear. First, an interface for the `UserRepository` class which will be implemented by the repository:

```
1 interface IUserRepository {
2
3     public function getAllUsers();
4
5     public function getUserById($id);
6
7     public function createOrUpdate($id = null);
8 }
```

Now we need to implement this interface in the `UserRepository` class using this:

```
1 // use the namespace if it has any, i.e.
2 // namespace Repositories\User;
3 // Same for IUserRepository interface
4
5 class UserRepository implements IUserRepository {
6
7     public function getAllUsers()
8     {
9         return User::all();
10    }
11
12    public function getUserById($id)
13    {
14        return User::find($id);
15    }
16
17    public function createOrUpdate($id = null)
18    {
19        if(is_null($id)) {
20            // create after validation
21            $user = new User;
22            $user->name = 'Sheikh Heera';
23            $user->email = 'me@yahoo.com';
24            $user->password = '123456';
25            return $user->save();
26        }
27        else {
28            // update after validation
29            $user = User::find($id);
30            $user->name = 'Sheikh Heera';
31            $user->email = 'me@yahoo.com';
32            $user->password = '123456';
33            return $user->save();
34        }
35    }
36 }
```

```
35     }
36
37 }
```

So, this repository will be used in the `UserController` but with the type hint of the interface that this class implemented instead of the concrete implementation (this `UserController` class):

```
1 class UserController extends BaseController {
2
3     protected $user = null;
4
5     // IUserRepository is the interface
6     public function __construct(IUserRepository $user)
7     {
8         $this->user = $user;
9     }
10
11    public function showUsers()
12    {
13        $users = $this->user->getAllUsers();
14        return View::make('user.index', compact('users'));
15    }
16
17    public function getUser($id)
18    {
19        $user = $this->user->getUserById($id);
20        return View::make('user.profile', compact('user'));
21    }
22
23    public function saveUser($id = null)
24    {
25        if($id) {
26            $this->user->createOrUpdate($id);
27        }
28        else {
29            $this->user->createOrUpdate();
30        }
31        // return redirect...
32    }
33
34 }
```

Now the `UserController` depends on the `IUserRepository` interface (abstract layer) instead of `UserRepository` (concrete class) and not tightly coupled with the `User` model so easily testable as well.

But still it's not ready to use by the application because an interface is not a class and the `IoC` container can't make an instance of this interface and it shouldn't but the container will try to do it because of the type hint and will fail, so it's necessary to tell the `IoC` container that, whenever the `UserController` class is getting instantiated, just pass an instance of the concrete `UserRepository` class which implemented the type hinted interface given in the constructor method. So, we need to bind the concrete class to the interface for the `IoC` container so it can supply the class to the controller and this could be done using this:

```
1 // IUserRepository interface, UserRepository class
2 App::bind('IUserRepository', 'UserRepository');
```

So, now the `IoC` is able to inject an instance of `UserRepository` class into `UserController` class whenever the `UserController` is initialized. If the repository and interface both inside a sub-directory and has namespace something like `namespace Repositories/User;` then during the binding it should be used this way:

```
1 // UserRepository interface, UserRepository class
2 App::bind('Repositories\\User\\IUserRepository', 'Repositories\\User\\UserRepository');
```

This could be placed in the `global.php` file or in another file (create one as `app_bindings.php`) where this code along with other code (for binding) could be placed and just include that file in the `global.php` file using `require '/app_bindings.php'` (assumed the file is in the same folder) or it's also possible to create a `Service Provider` so Laravel will register this at the boot up process of the framework. To create a service provider, create a file in the same folder and save it as `UserServiceProvider.php` and use paste this code:

```
1 <?php namespace Repositories\\User;
2
3 use Illuminate\\Support\\ServiceProvider;
4
5 class UserServiceProvider extends ServiceProvider {
6
7     public function register()
8     {
9         $this->app::bind('Repositories\\User\\IUserRepository', 'Repositories\\User\\UserRepository');
10    }
11 }
```

Then in the `providers` array in the `app/config/app.php` file, add this at the last:

```
1 'providers' => array(
2     // more entries ...
3     'Repositories\\User\\UserServiceProvider'
4 )
```

Using a service provider is not necessary but a good way to organize things. So, by using an interface we just not created a contract but also the detail of the implementation is hidden (abstracted) from the controller so it doesn't know anything about the class is being used behind the scene and this is a plus point. According to the design principles (LSP in **SOLID**) "objects in a program should be replaceable with instances of their `subtypes` without altering the correctness of that program" which also relates to **Design by contract** and these rules or principles are obviously for good reason and one of the good reasons is that, now it's possible to change the implementation of the concrete class of `UserRepository` with a different one which also implements the `IUserRepository` interface and hence the controller will work normally because it doesn't know what is the concrete class is being used for that interface, it's totally ignorant about that and that's why we can also test our controller without using a real database because we can pass a mock object instead of original `UserRepository` class, only we have to implement the same interface and the controller will think it's using the same thing.

So, for example to use a different implementation of the interface we may implement another repository by implementing the `IUserRepository` interface and just can change the class/repository binding for the container and we don't need to touch our controller code. FOR example if we implement something like this:

```
1 <?php namespace Repositories\User;
2
3 class MongoUserRepository implements IUserRepository {
4
5     public function getAllUsers()
6     {
7         // get and return all user mongodb
8     }
9
10    public function getUserById($id)
11    {
12        // get and return the user from mongodb
13    }
14
15    public function createOrUpdate($id = null)
16    {
17        // add a new entry in mongodb
18    }
19
20 }
```

Now, in the service provider we only need to change the binding:

```
1 $this->app::bind('Repositories\\User\\IUserRepository', 'Repositories\\User\\MongoUserRepository');
```

That's it, this is now more maintainable and easily testable because of the use of abstract layer (interface) instead of the concrete type.

Another thing we can do in our repository class to use the methods of `Eloquent` (User) model, for example, from our `Usercontroller` class we can't use `UserRepository::with()` unless we declare that method in the class but it's possible to use those non-existing methods (in the repository) using `__call()` magic method like this (also a constructor):

```
1 class UserRepository implements IUserRepository {
2
3     protected $user = null;
4
5     public function __construct(\User $user)
6     {
7         $this->user = $user;
8     }
9
10    public function getAllUsers()
11    {
12        return $this->user->all();
13    }
14
15    public function __call($method, $args)
16    {
17        return call_user_func_array([$this->user, $method], $args);
18    }
19
20 }
```

So, now we can use all the methods of `User` model from the `UserController` (if required) as if those methods are declared in the `UserRepository` while they are not. For example:

```
1 class UserController extends BaseController {
2
3     protected $user = null;
4
5     public function __construct(IUserRepository $user)
6     {
7         $this->user = $user;
8     }
9
10    // Can access the methods of User model
11    public function getUserWithPostsComments()
12    {
13        $user = $this->user->whereEmail('me@yahoo.com')
14                    ->with('posts')
15                    ->with('comments')
16                    ->get();
17    }
18 }
```

This may not required when we use the repository pattern but it's just an idea to use model's methods easily.

The Repository pattern is not limited to `Laravel` or `PHP` but it's a widely used software architecture or design pattern used by developers using various languages.

So, finally, it's not important that you must follow the rules and principles in every project but it's a good practice but if you don't use it, it'll still work and it's not a sin, you should decide what to do and how to do it and by following some rules blindly doesn't makes any sense, it may create problems if applied inappropriately, so why not use your sense without blindly following some pro, some one said it's good practice so I'll do this, if this is the case then it's not good at all. I always like to find alternatives and think as `what if`, Oops! it's too much...

157 Comments

Heera.IT

 Login ▾

 Recommend 7

 Tweet

 Share

Sort by Newest ▾



Join the discussion...

LOG IN WITH



OR SIGN UP WITH DISQUS 

Name



nullReference • a year ago

I've been researching the best way to go about implementing a repository pattern in laravel and this REALLY helped me to put all the pieces together! Greatly Appreciated! :)

^ | v • Reply • Share ›



saurabh kamble → nullReference • a month ago

How can we going unit testing of repositories using database any ideas ?

^ | v • Reply • Share ›



Sheikh Heera Moderator → saurabh kamble • a month ago

It's a broad question but this may help: <https://www.google.com/sear...>

1 ^ | v • Reply • Share ›



Tuan Pham • a year ago

Thanks for the detailed post.

I have a question. Since your repository is returning the Eloquent Models, if you decide to change to use **MongoUserRepository**, how do you implement **with()**, **whereEmail()**, **get()** and other Laravel Eloquent magic methods? Your repository is not really is a repository, it is just the new place of the old code from your controller.

3 ^ | v • Reply • Share ›



Sheikh Heera Moderator → Tuan Pham • a year ago

In that case you'll need to use a different repository that implements the same interface and you've to use a mongodb orm implementation of eloquent. Btw, this is not a rule but one way of doing it using repository patter and in most cases you might don't need it but in the context of Laravel, it might work. I don't use repository all the time but rarely. So, don't fall into that trap without understating the need.

15 ^ | v • Reply • Share ›



Tuan Pham → Sheikh Heera • a year ago

Please consider to remove the idea

using `__call()` magic method

, because many programmers will see it is very convenient and apply it without thoughts...

That idea also break your repository contract/interface, break unit test...

It's bad and known as anti pattern because the controller has hard dependencies on the User model and hence it's tightly coupled to each other and that's why it's hard to test (unit testing)

why we can also test our controller without using a real database because we can pass a mock object instead of original UserRepository class, only we have to implement the same interface and the controller will think it's using the same thing

2 ^ | v • Reply • Share ›

[Show more replies](#)



sicksadworld • a year ago

thank you

1 ^ | v • Reply • Share ›



Sheikh Heera Moderator → sickсадworld • a year ago

Most welcome :-)

3 ^ | v • Reply • Share ›



Surbhi Jain • a year ago

Assuming that introduction of Laravel developer processing with an **laravel development services**

^ | v • Reply • Share ›



Hiền Lê • 2 years ago

I tried many times with other guides which are similare to your post but not successful :(

In UserController, it always show error:

Call to a member function all() on null, which means \$this->user is null. Why your UserController doesn't have instance \$user, where do you get it? Please help me. Thanks.

^ | v • Reply • Share ›



Dinh Nguyen → Hiền Lê • a year ago

Hien,

There is a mistake in UserController. \$user must be declared in it like below:

```
class UserController extends BaseController {
    private $user;

    // IUserRepository is the interface
    public function __construct(IUserRepository $user)
    {
        $this->user = $user;
    }
}
```

^ | v • Reply • Share ›



Sheikh Heera Moderator → Dinh Nguyen • a year ago

Thanks, but it'll work, anyways, by default it'll be a public property but I like to declare protected properties.

3 ^ | v • Reply • Share ›



Sheikh Heera Moderator → Hiền Lê • 2 years ago

Do you have bindings in place `App::bind('Repositories\\User\\IUserRepository', 'Repositories\\User\\UserRepository');`?

^ | v • Reply • Share ›



Manish Shrestha • 2 years ago

Wonderful!

1 ^ | v • Reply • Share ›



Sheikh Heera Moderator → Manish Shrestha • 2 years ago

Thanks **@Manish Shrestha** , it's too old tho :-)

^ | v • Reply • Share ›



Manish Shrestha → Sheikh Heera • 2 years ago

Is it? I've just started to research on Repository Pattern to build my new SaaS project. Am I missing anything new that has already been introduced to the market?

^ | v • Reply • Share ›

[Show more replies](#)



Dmitriy Doronin • 2 years ago

Great article! I'm really understood why we should be use repository pattern.

1 ^ | v • Reply • Share ›



mmosw • 2 years ago

Another thing we can do in our repository class to use the methods of Eloquent (User) model, for example, from our UserController class we can't use UserRepository::with() unless we declare that method in the class but it's possible to use those non-existing methods (in the repository) using __call() magic method like this (also a constructor):...

Why do you suggest doing this, it make your repository and controller fully coupled with Eloquent, which only support for certain SQL database? Then how do you implement the MongoUserRepository? Writing new Eloquent model that support mongodb? That is not easy.

I think you should write this function in your repository rather than in controller.

```
// Can access the methods of User model
public function getUserWithPostsComments()
{
    $user = $this->user->whereEmail('me@yahoo.com')
->with('posts')
->with('comments')
->get();
}
```

[see more](#)

1 ^ | v • Reply • Share ›



1p0 • 3 years ago

great article. My 2 cents... what about code navigability in IDEs like phpStorm, eclipse PDT, etc? if the controller does link to non injected classes without interfaces, as a direct dependency, you can control+click/cmd+click a method, and the IDE takes you directly to it's implementation and you can repeat this several times. Now the abstraction does break navigability which is a problem in projects that are old or big or unknown, specially for people that just jump into the team. Yes you can use type hinting in comments to say

```
/** @var UserModel $object */
```

```
protected $user;
```

And so on, which is a "soft comment default type hinting sort of a workaround".

But the most important thing to keep in mind for me on software architecture, is that it should work for us, and not the other way around. And that's where your comment of "you need to evaluate in a per project basis" comes handy.

So, great article, and let's hope IDEs continue evolving to solve this problems for us :D

3 ^ | v • Reply • Share ›



karim • 4 years ago

Thank you for tutorial,

i get confused about MongoUserRepository class, after biding, how could i inject this class into the UserController class. for example suppose the UserRepository and MongoUserRepository classes are required to be passed to the UserController, is the following code will be used to achieve this? How the IoC will know which concrete class to load since i pass to it IUserRepository interface?

```
class UserController extends BaseController {

public function __construct(IUserRepository $user, IUserRepository $mongo_user)
{
    $this->user = $user;
    $this->mongo_user = $mongo_user;
}

}
```

Thanks

1 ^ | v • Reply • Share ›



Sheikh Heera Moderator → karim • 4 years ago

Welcome @karim :-)

Anyways, Basically you would bind only one implementation with the interface. `Laravel 5` has contextual bindings available but that's not available in `Laravel 4`. So, make sure you bind two different interfaces for both Repository or use a different approach, for example, you may use a `trait` or inheritance mechanism to add extra functionality into an existing `UserRepository`. Hope it helps. Check the documentation <http://laravel.com/docs/4.2...> for more.

^ | v • Reply • Share ›



Divo • 4 years ago

Gracias for this awesome article. Just from your introduction I see you have good understanding of software development principles and so I just had to come and comment even before reading the whole article. `Laravel` is awesome and make me enjoy development all the more. Keep sharing. Thank you.

1 ^ | v • Reply • Share ›



Sheikh Heera Moderator → Divo • 4 years ago

Thanks @@Divo. Glad to know that you enjoy coding with `Laravel`. I just try to learn and share. All the best :-)

^ | v • Reply • Share ›



Divo → Sheikh Heera • 4 years ago



Sheikh Heera • 4 years ago

Works well in Laravel 5 with a few modifications. Now suppose I have 2 or more interfaces and repositories with similar methods. What can I do to address such duplication so I don't find my self implementing the same methods in separate repositories?

1 ^ | v • Reply • Share ›

[Show more replies](#)



mak • 4 years ago

Thank you so much my friend.sheikh Heera You saved my life by sharing this wonderful concept about laravel.

You did well.

Thank you so much again.

1 ^ | v • Reply • Share ›



Sheikh Heera **Moderator** → mak • 4 years ago

You are welcome **@mak** :-)

^ | v • Reply • Share ›



This comment is awaiting moderation. [Show comment.](#)



Sheikh Heera **Moderator** → Николай Йоцов • 4 years ago

Thanks for your participation, anyways, regarding `Generic`

repository, yes, that's possible and sometimes it's much better to use one `generic` class/repo to handle the common actions using different models. For example, check this `Generic` handler:

```
class crudHandler {

    protected $model = null;

    public function setModel(Eloquent $model)
    {
        $this->model = $model;
    }

    public function getModel()
    {
        return $this->model;
    }
}
```

[see more](#)

^ | v • Reply • Share ›



Fahim Durrani • 4 years ago

What a thoroughly descriptive article! Awesome work.

1 ^ | v • Reply • Share ›



Sheikh Heera **Moderator** → Fahim Durrani • 4 years ago



Thank you **@Fahim Durrani** :-)

^ | v • Reply • Share ›



Rehana Chowdhury • 4 years ago

well

1 ^ | v • Reply • Share ›



Chris • 4 years ago

Sheikh, Why is it hard to unit test a tightly coupled Controller -> Model?

Why is it easier when a Repository is in between?

"Repository" sounds like a "helper/library" + "contract"

Can't understand how it makes testing any easier. :-/

^ | v • Reply • Share ›



waterloomatt → Chris • 4 years ago

One of the main reasons is during testing you usually want to use mock objects which don't actually hit or modify the DB. You want to be able to run the test over and over again without worrying about duplicate constraints or contaminating your data. Ex. you'd switch out User for MockUser which also implements the IUserRepository and your controller would be none the wiser.

2 ^ | v • Reply • Share ›



Sheikh Heera **Moderator** → waterloomatt • 4 years ago

Btw, the repository also abstract away the persitant layer and useful for domain driven development. but this is not always the case. Here I only demonstrate the use exclusively for Laravel. Otherwise a repository should return native data types (array) instead of Eloquent/collections.

^ | v • Reply • Share ›



Sheikh Heera **Moderator** → Chris • 4 years ago

Probably you didn't read the post properly, hard coded classes are dependencies to a top level module so it's hard to swap at run time.

1 ^ | v • Reply • Share ›



Chris → Sheikh Heera • 4 years ago

Thanks for the reply. I believe I need more experience testing to truly understand. :-)

1 ^ | v • Reply • Share ›



Miguel Mendez • 4 years ago

By returning an instance of User on getUserById isn't your controller still coupled to the Model Class? It is fine using an interface but the return should be also an interface, right? Correct me if I'm wrong.

1 ^ | v • Reply • Share ›



Sheikh Heera **Moderator** → Miguel Mendez • 4 years ago

The Interface is switchable and that's why you may change the implementation for a different object, for a generic result you may use another layer before returning to the client/consumer

client/consumer.

^ | v • Reply • Share ›



Miguel Mendez → Sheikh Heera • 4 years ago

That's what I thought, somehow you could agree on a defined Domain Model not tied to Eloquent that can be returned, like a POJO in Java, that way it does not matter how the repository implements fetching the data as long as it returns the defined Domain Model. Thanks for a fine article!

^ | v • Reply • Share ›



ahmed • 4 years ago

Hello Sheikh Heera! I finally found what i'm searching for and finally understood how to use Repository pattern and why to use it i'm not using laravel but you really explain it very well I want to thank you alot about this article thanks man.

^ | v • Reply • Share ›



Sheikh Heera **Moderator** → ahmed • 4 years ago

Glad it was helpful, you are most welcome :-)

^ | v • Reply • Share ›



Basa Gabi • 4 years ago

Hello Sheikh Heera! I really find this article useful though I have a question. Is it possible for multiple repositories to implement the same interface? Base on your example given; class UserRepository implements IUserRepository in which class UserController uses IUserRepository via dependency injection, right? Now my question is:

1. Could the interface be implemented on other repositories? Ex: I have GenderRepository that implements IUserRepository

2. If #1 is possible, how would I do that if the interface is being used to be injected on the construct method of the UserController? How would we know what repository it is using?

Thanks a lot!

1 ^ | v • Reply • Share ›



Sheikh Heera **Moderator** → Basa Gabi • 4 years ago

Welcome! Yes, you can implement more than repository using same contract/interface but there is no way (unless you find out) to do this in Laravel-4 but in 5, you may use <http://laravel.com/docs/5.0...> for this. Probably, you may use a setter injection instead of constructor injection in case of version 4 and manually set the repository at runtime.

^ | v • Reply • Share ›



Basa Gabi → Sheikh Heera • 4 years ago

Thanks you for the response! Where would I place the `$this->app->when()`...? On the service provider under the register method?

Given the said code on the docs, if I translate it, is this correct?

```
$this->app->when('App\Controllers\WelcomeController')
->needs('App\Repositories\Interfaces\MyInterface')
->give('App\Repositories\UserRepository');
```

Thanks so much!

1 ^ | v • Reply • Share ›

[Show more replies](#)



Kyslik • 4 years ago

Hello Sheikh Heera, thank you for this article in fact I've read many similar articles to this one and none answers few questions I have or many people might have.

1. relations the most important thing Laravel provides out of box no example to be found - I am asking for an example lets say M to N relation - while using repository pattern of course.
2. How can I pass messages between lets say UserRepository and RoleRepository. By message I mean how do I tie the two together. How does one use instance of UserRepository inside RoleRepository and vice versa.
3. what about packages that are tied directly to the Eloquent? How do I force packages made by 3rd party use my Repos? - As I think about this it is invalid question, too specific and impossible to answer. So see 3b.
- 3b. Is it possible to force L5 use my UserRepository instead Eloquent model? While not to reinvent the wheel - <http://laravel.com/docs/5.0...>
4. How can I use scopes with repositories?

Thank you.

1 ^ | v • Reply • Share ›



Sheikh Heera Moderator ➔ Kyslik • 4 years ago

Q-1. A: <http://laravel.com/docs/5.0...>

Q-2 A: Instead of using two repositories use one Repository and use models as it's dependencies, i.e:

```
use namespaces/classes....

class UserRepository implements UserInterface {

    public function __construct(RoleIModel $role, User $user)
    {
        $this->role = $role;
        $this->user = $user;
    }
}
```

If you need to use authenticated user then you may use `Request::user()`, use Request contract instead of `Request Facade`

[see more](#)

1 ^ | v • Reply • Share ›



Sheikh Heera Moderator ➔ Kyslik • 4 years ago

Hi @Kyslik, sorry for the delay but unfortunately I didn't understand what you asked, can you please re-phrase your question by spiting each questions? I'll try to answer ;-)

1 ^ | v • Reply • Share ›

^ | v • Reply • Share ›



Kyslik → Sheikh Heera • 4 years ago

Please see edits if question is more understandable :). Thank you.

^ | v • Reply • Share ›



Safoor Safdar • 4 years ago

Could you please update according to laravel-5.

1 ^ | v • Reply • Share ›



Sheikh Heera **Moderator** → Safoor Safdar • 4 years ago

I'll try it very soon, just need a gap from my current work (for a client) but it works because I've upgraded one project (currently working) from this to new version and it just works without any errors but I've modified the code a bit for more flexibility without the new features the old one (this example) works with latest official version, so may use it same way, but I'll upgrade it soon with a bootstrap 3 template and production ready code so taking time. All the best :-)

1 ^ | v • Reply • Share ›

Load more comments

ALSO ON HEERA.IT

Laravel Tips – Update and Persist Config on Runtime

11 comments • 5 years ago



Sheikh Heera — You are most welcome :-)
Avatar

Taste of JavaScript in Php

2 comments • 6 years ago



MD.Mahbub Helal — I like the way you think
Avatar); Thanks for the article.

Laravel-5.0 And Routing

4 comments • 5 years ago



Sito Demmers — Ok Sheikh! keep up the good
Avatarwork :-) and thanks again, I found out about it,
it's really simple and great this new routing...

Laravel – 5.1 ACL Using Middleware

162 comments • 4 years ago



Sheikh Heera — This is designed to protect
Avatarroutes so you have to place permissions in
route.



Subscribe



Add Disqus to your site



Disqus' Privacy Policy

DISQUS

Newsletter


Subscribe to my email newsletter for useful tips and valuable Resources.

Email Address



Me on StackOverflow



 **The Alpha**
114,351
● 20 ● 220 ● 249

Recent Posts

Hidden Gems of WordPress – Part 2

Hidden Gems of WordPress – Part 1

Ninja Tables Dynamic Data Filter

Laravel – Nested Relationship Revised

AdonisJs – A Laravel-ish Node Framework

Categories

Php

Javascript

Plugins

Miscellaneous

Random Ideas

Mobile Application