**Gates and Policies in Laravel**

Today, we're going to discuss the authorization system of the Laravel web framework. The Laravel framework implements authorization in the form of gates and policies. After an introduction to gates and policies, I'll demonstrate the concepts by implementing a custom example.

I assume that you're already aware of the built-in Laravel authentication system as that's something essential in order to understand the concept of authorization. Obviously, the authorization system works in conjunction with the authentication system in order to identify the legitimate user session.

If you're not aware of the Laravel authentication system, I would highly recommend going through the [official documentation](https://laravel.com/docs/5.5/authentication), which provides you with hands-on insight into the subject.

**Laravel's Approach to Authorization**

By now, you should already know that the Laravel authorization system comes in two flavors—gates and policies. Although it may sound like a complicated affair, I would say it's pretty easy to implement it once you get the hang of it!

Gates allow you to define an authorization rule using a simple closure-based approach. In other words, when you want to authorize an action that's not related to any specific model, the gate is the perfect place to implement that logic.

Let's have a quick look at what gate-based authorization looks like:

|  |  |
| --- | --- |
| 4  5 | Gate::define('update-post', function ($user, $post) {    return $user->id == $post->user\_id;  }); |

The above snippet defines the authorization rule update-post that you could call from anywhere in your application.

On the other hand, you should use policies when you want to group the authorization logic of any model. For example, let's say you have a Post model in your application, and you want to authorize the CRUD actions of that model. In that case, it's the policy that you need to implement.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class PostPolicy  {    public function view(User $user, Post $post) {}    public function create(User $user) {}    public function update(User $user, Post $post) {}    public function delete(User $user, Post $post) {}  } |

As you can see, it's a pretty simple policy class that defines the authorization for the CRUD actions of the Post model.

So that was an introduction to gates and policies in Laravel. From the next section onwards, we'll go through a practical demonstration of each element.

**Gates**

In this section, we'll see a real-world example to understand the concept of gates.

More often than not, you end up looking at the Laravel service provider when you need to register a component or a service. Following that convention, let's go ahead and define our custom gate in the app/Providers/AuthServiceProvider.php as shown in the following snippet.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <?php  namespace App\Providers;    use Illuminate\Support\Facades\Gate;  use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;  use Illuminate\Http\Request;    class AuthServiceProvider extends ServiceProvider  {    /\*\*     \* The policy mappings for the application.     \*     \* @var array     \*/    protected $policies = [      'App\Model' => 'App\Policies\ModelPolicy',    ];      /\*\*     \* Register any authentication / authorization services.     \*     \* @return void     \*/    public function boot()    {      $this->registerPolicies();        Gate::define('update-post', function ($user, $post) {        return $user->id == $post->user\_id;      });    }  } |

In the boot method, we've defined our custom gate:

|  |  |
| --- | --- |
| 1  2  3 | Gate::define('update-post', function ($user, $post) {    return $user->id == $post->user\_id;  }); |

While defining a gate, it takes a closure that returns either TRUE or FALSE based on the authorization logic that's defined in the gate definition. Apart from the closure function, there are other ways you could define gates.

For example, the following gate definition calls the controller action instead of the closure function.

|  |  |
| --- | --- |
| 1 | Gate::define('update-post', 'ControllerName@MethodName'); |

Now, let's go ahead and add a custom route so that we can go through a demonstration of how gate-based authorization works. In the routes file routes/web.php, let's add the following route.

|  |  |
| --- | --- |
| 1 | Route::get('service/post/gate', 'PostController@gate'); |

Let's create an associated controller file app/Http/Controllers/PostController.php as well.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <?php  namespace App\Http\Controllers;    use App\Http\Controllers\Controller;  use App\Post;  use Illuminate\Support\Facades\Gate;    class PostController extends Controller  {    /\* Make sure you don't user Gate and Policy altogether for the same Model/Resource \*/    public function gate()    {      $post = Post::find(1);        if (Gate::allows('update-post', $post)) {        echo 'Allowed';      } else {        echo 'Not Allowed';      }        exit;    }  } |

In most cases, you'll end up using either the allows or denies method of the Gate facade to authorize a certain action. In our example above, we've used the allows method to check if the current user is able to perform the update-post action.

Users with sharp eyes would have noticed that we've only passed the second argument $post to the closure. The first argument, the current logged-in user, is automatically injected by the Gate facade.

So that's how you're supposed to use gates to authorize actions in your Laravel application. The next section is all about how to use policies, should you wish to implement authorization for your models.

**Policies**

As we discussed earlier, when you want to logically group your authorization actions for any particular model or resource, it's the policy you're looking for.

In this section, we'll create a policy for the Post model that will be used to authorize all the CRUD actions. I assume that you've already implemented the Post model in your application; otherwise, something similar will do.

The Laravel artisan command is your best friend when it comes to creating stubbed code. You can use the following artisan command to create a policy for the Post model.

|  |  |
| --- | --- |
| 1 | $php artisan make:policy PostPolicy --model=Post |

As you can see, we've supplied the --model=Post argument so that it creates all the CRUD methods. In the absence of that, it'll create a blank Policy class. You can locate the newly created Policy class at app/Policies/PostPolicy.php.

Let's replace it with the following code.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | <?php  namespace App\Policies;    use App\User;  use App\Post;  use Illuminate\Auth\Access\HandlesAuthorization;    class PostPolicy  {    use HandlesAuthorization;      /\*\*     \* Determine whether the user can view the post.     \*     \* @param  \App\User  $user     \* @param  \App\Post  $post     \* @return mixed     \*/    public function view(User $user, Post $post)    {      return TRUE;    }      /\*\*     \* Determine whether the user can create posts.     \*     \* @param  \App\User  $user     \* @return mixed     \*/    public function create(User $user)    {      return $user->id > 0;    }      /\*\*     \* Determine whether the user can update the post.     \*     \* @param  \App\User  $user     \* @param  \App\Post  $post     \* @return mixed     \*/    public function update(User $user, Post $post)    {      return $user->id == $post->user\_id;    }      /\*\*     \* Determine whether the user can delete the post.     \*     \* @param  \App\User  $user     \* @param  \App\Post  $post     \* @return mixed     \*/    public function delete(User $user, Post $post)    {      return $user->id == $post->user\_id;    }  } |

To be able to use our Policy class, we need to register it using the Laravel service provider as shown in the following snippet.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <?php  namespace App\Providers;    use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;  use Illuminate\Http\Request;  use App\Post;  use App\Policies\PostPolicy;    class AuthServiceProvider extends ServiceProvider  {    /\*\*     \* The policy mappings for the application.     \*     \* @var array     \*/    protected $policies = [      'App\Model' => 'App\Policies\ModelPolicy',      Post::class => PostPolicy::class    ];      /\*\*     \* Register any authentication / authorization services.     \*     \* @return void     \*/    public function boot()    {      $this->registerPolicies();    }  } |

We've added the mapping of our Policy in the $policies property. It tells Laravel to call the corresponding policy method to authorize the CRUD action.

You also need to register the policies using the registerPolicies method, as we've done in the boot method.

Moving further, let's create a couple of custom routes in the routes/web.php file so that we can test our Policy methods there.

|  |  |
| --- | --- |
| 1  2  3  4 | Route::get('service/post/view', 'PostController@view');  Route::get('service/post/create', 'PostController@create');  Route::get('service/post/update', 'PostController@update');  Route::get('service/post/delete', 'PostController@delete'); |

Finally, let's create an associated controller at app/Http/Controllers/PostController.php.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | <?php  namespace App\Http\Controllers;    use App\Http\Controllers\Controller;  use App\Post;  use Illuminate\Support\Facades\Auth;    class PostController extends Controller  {    public function view()    {      // get current logged in user      $user = Auth::user();        // load post      $post = Post::find(1);        if ($user->can('view', $post)) {        echo "Current logged in user is allowed to update the Post: {$post->id}";      } else {        echo 'Not Authorized.';      }    }      public function create()    {      // get current logged in user      $user = Auth::user();        if ($user->can('create', Post::class)) {        echo 'Current logged in user is allowed to create new posts.';      } else {        echo 'Not Authorized';      }        exit;    }      public function update()    {      // get current logged in user      $user = Auth::user();        // load post      $post = Post::find(1);        if ($user->can('update', $post)) {        echo "Current logged in user is allowed to update the Post: {$post->id}";      } else {        echo 'Not Authorized.';      }    }      public function delete()    {      // get current logged in user      $user = Auth::user();        // load post      $post = Post::find(1);        if ($user->can('delete', $post)) {        echo "Current logged in user is allowed to delete the Post: {$post->id}";      } else {        echo 'Not Authorized.';      }    }  } |

There are different ways you could authorize your actions using Policies. In our example above, we’ve used the User model to authorize our Post model actions.

The User model provides two useful methods for authorization purposes—can and cant. The can method is used to check if the current user is able to execute a certain action. And the counterpart of the can method, the cant method, is used to determine the inability of the action execution.

Let’s grab the snippet of the view method from the controller to see what exactly it does.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14 | public function view()  {    // get current logged in user    $user = Auth::user();      // load post    $post = Post::find(1);      if ($user->can('view', $post)) {      echo "Current logged in user is allowed to update the Post: {$post->id}";    } else {      echo 'Not Authorized.';    }  } |

Firstly, we load the currently logged-in user, which gives us the object of the User model. Next, we load an example post using the Post model.

Moving ahead, we’ve used the can method of the User model to authorize the view action of the Post model. The first argument of the can method is the action name that you want to authorize, and the second argument is the model object that you want to get authorized against.

That was a demonstration of how to use the User model to authorize the actions using policies. Alternatively, you could use the *Controller Helper* as well, if you’re in the controller while authorizing a certain action.

|  |  |
| --- | --- |
| 1  2  3 | …  $this->authorize('view', $post);  … |

As you can see, you don’t need to load the User model if you use the Controller Helper.

So that was the concept of policies at your disposal, and it’s really handy while authorizing a model or a resource as it allows you to group the authorization logic in one place.

Just make sure that you don’t use gates and policies altogether for the same actions of the Model, otherwise it’ll create issues. That’s it from my side for today, and I’ll call it a day!

**Conclusion**

Today, it was Laravel authorization that took the center stage in my article. At the beginning of the article, I introduced the main elements of Laravel authorization, gates and policies.

Following that, we went through creating our custom gate and policy to see how it works in the real world. I hope you’ve enjoyed the article and learned something useful in the context of Laravel.

For those of you who are either just getting started with Laravel or looking to expand your knowledge, site, or application with extensions, we have a variety of things you can study on [Envato Market](http://codecanyon.net/search?referrer=search&sort=&term=Laravel&utf8=%E2%9C%93&view=list).

Top of Form

Bottom of Form