Authorization

# Introduction

In addition to providing authentication services out of the box, Laravel also provides a simple way to authorize user actions against a given resource. Like authentication, Laravel's approach to authorization is simple, and there are two primary ways of authorizing actions: gates and policies.

Think of gates and policies like routes and controllers. Gates provide a simple, Closure based approach to authorization while policies, like controllers, group their logic around a particular model or resource. We'll explore gates first and then examine policies.

You do not need to choose between exclusively using gates or exclusively using policies when building an application. Most applications will most likely contain a mixture of gates and policies, and that is perfectly fine! Gates are most applicable to actions which are not related to any model or resource, such as viewing an administrator dashboard. In contrast, policies should be used when you wish to authorize an action for a particular model or resource.

Gates

Writing Gates

Gates are Closures that determine if a user is authorized to perform a given action and are typically defined in the `App\Providers\AuthServiceProvider` class using the `Gate` facade. Gates always receive a user instance as their first argument, and may optionally receive additional arguments such as a relevant Eloquent model:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('edit-settings', function ($user) {
        return $user->isAdmin;
    });

    Gate::define('update-post', function ($user, $post) {
        return $user->id === $post->user_id;
    });
}
```

Gates may also be defined using a `Class@method` style callback string, like controllers:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', 'App\Policies\PostPolicy@update');
}
```

Authorizing Actions

To authorize an action using gates, you should use the `allows` or `denies` methods. Note that you are not required to pass the currently authenticated user to these methods. Laravel will automatically take care of passing the user into the gate Closure:

```
if (Gate::allows('edit-settings')) {
    // The current user can edit settings
}

if (Gate::allows('update-post', $post)) {
    // The current user can update the post...
}

if (Gate::denies('update-post', $post)) {
    // The current user can't update the post...
}
```

If you would like to determine if a particular user is authorized to perform an action, you may use the `forUser` method on the `Gate` facade:

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // The user can update the post...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
    // The user can't update the post...
}
```

You may authorize multiple actions at a time with the `any` or `none` methods:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // The user can update or delete the post
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // The user cannot update or delete the post
}
```

Authorizing Or Throwing Exceptions

If you would like to attempt to authorize an action and automatically throw an `Illuminate\Auth\Access\AuthorizationException` if the user is not allowed to perform the given action, you may use the `Gate::authorize` method. Instances of `AuthorizationException` are automatically converted to `403` HTTP response:

```
Gate::authorize('update-post', $post);

// The action is authorized...
```

Supplying Additional Context

The gate methods for authorizing abilities (`allows`, `denies`, `check`, `any`, `none`, `authorize`, `can`, `cannot`) and the authorization Blade directives (`@can`, `@cannot`, `@canany`) can receive an array as the second argument. These array elements are passed as parameters to gate, and can be used for additional context when making authorization decisions:

```
Gate::define('create-post', function ($user, $category, $extraFlag) {
    return $category->group > 3 && $extraFlag === true;
});

if (Gate::check('create-post', [$category, $extraFlag])) {
    // The user can create the post...
}
```

Gate Responses

So far, we have only examined gates that return simple boolean values. However, sometimes you may wish to return a more detail response, including an error message. To do so, you may return a `Illuminate\Auth\Access\Response` from your gate:

```
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function ($user) {
    return $user->isAdmin
                ? Response::allow()
                : Response::deny('You must be a super administrator.');
});
```

When returning an authorization response from your gate, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```
$response = Gate::inspect('edit-settings', $post);

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

Of course, when using the `Gate::authorize` method to throw an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```
Gate::authorize('edit-settings', $post);

// The action is authorized...
```

Intercepting Gate Checks

Sometimes, you may wish to grant all abilities to a specific user. You may use the `before` method to define a callback that is run before all other authorization checks:

```
Gate::before(function ($user, $ability) {
    if ($user->isSuperAdmin()) {
        return true;
    }
});
```

If the `before` callback returns a non-null result that result will be considered the result of the check.

You may use the `after` method to define a callback to be executed after all other authorization checks:

```
Gate::after(function ($user, $ability, $result, $arguments) {
    if ($user->isSuperAdmin()) {
        return true;
    }
});
```

Similar to the `before` check, if the `after` callback returns a non-null result that result will be considered the result of the check.

Creating Policies

Generating Policies

Policies are classes that organize authorization logic around a particular model or resource. For example, if your application is a blog, you may have a `Post` model and a corresponding `PostPolicy` to authorize user actions such as creating or updating posts.

You may generate a policy using the `make:policy` artisan command. The generated policy will be placed in the `app/Policies` directory. If this directory does not exist in your application, Laravel will create it for you:

```
php artisan make:policy PostPolicy
```

The `make:policy` command will generate an empty policy class. If you would like to generate a class with the basic "CRUD" policy methods already included in the class, you may specify a `--model` when executing the command:

```
php artisan make:policy PostPolicy --model=Post
```

All policies are resolved via the Laravel service container, allowing you to type-hint any needed dependencies in the policy's constructor to have them automatically injected.

Registering Policies

Once the policy exists, it needs to be registered. The `AuthServiceProvider` included with fresh Laravel applications contains a `policies` property which maps your Eloquent models to their corresponding policies. Registering a policy will instruct Laravel which policy to utilize when authorizing actions against a given model:

```php
<?php

namespace App\Providers;

use App\Policies\PostPolicy;
use App\Post;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider
use Illuminate\Support\Facades\Gate;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        //
    }
}
```

Policy Auto-Discovery

Instead of manually registering model policies, Laravel can auto-discover policies as long as the model and policy follow standard Laravel naming conventions. Specifically, the policies must be in a `Policies` directory below the directory that contains the models. So, for example, the models may be placed in the `app` directory while the policies may be placed in the `app/Policies` directory. In addition, the policy name must match the model name and have a `Policy` suffix. So, a `User` model would correspond to a `UserPolicy` class.

If you would like to provide your own policy discovery logic, you may register a custom callback using the `Gate::guessPolicyNamesUsing` method. Typically, this method should be called from the `boot` method of your application's `AuthServiceProvider`:

```php
use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function ($modelClass) {
    // return policy class name...
});
```

Any policies that are explicitly mapped in your `AuthServiceProvider` will take precedence over any potential auto-discovered policies.

Writing Policies

Policy Methods

Once the policy has been registered, you may add methods for each action it authorizes. For example, let's define an `update` method on our `PostPolicy` which determines if a given `User` can update a given `Post` instance.

The `update` method will receive a `User` and a `Post` instance as its arguments, and should return `true` or `false` indicating whether the user is authorized to update the given `Post`. So, for this example, let's verify that the user's `id` matches the `user_id` on the post:

```php
<?php

namespace App\Policies;

use App\Post;
use App\User;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\User  $user
     * @param  \App\Post  $post
     * @return bool
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

You may continue to define additional methods on the policy as needed for the various actions it authorizes. For example, you might define `view` or `delete` methods to authorize various `Post` actions, but remember you are free to give your policy methods any name you like.

If you used the `--model` option when generating your policy via the Artisan console, it will already contain methods for the `view`, `create`, `update`, `delete`, `restore`, and `forceDelete` actions.

Policy Responses

So far, we have only examined policy methods that return simple boolean values. However, sometimes you may wish to return a more detail response, including an error message. To do so, you may return a `Illuminate\Auth\Access\Response` from your policy method:

```php
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.
 *
 * @param  \App\User  $user
 * @param  \App\Post  $post
 * @return bool
 */
public function update(User $user, Post $post)
{
    return $user->id === $post->user_id
                ? Response::allow()
                : Response::deny('You do not own this post.');
}
```

When returning an authorization response from your policy, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```php
$response = Gate::inspect('update', $post);

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

Of course, when using the `Gate::authorize` method to throw an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```php
Gate::authorize('update', $post);

// The action is authorized...
```

Methods Without Models

Some policy methods only receive the currently authenticated user and not an instance of the model they authorize. This situation is most common when authorizing `create` actions. For example, if you are creating a blog, you may wish to check if a user is authorized to create any posts at all.

When defining policy methods that will not receive a model instance, such as a `create` method, it will not receive a model instance. Instead, you should define the method as only expecting the authenticated user:

```
/**
 * Determine if the given user can create posts.
 *
 * @param  \App\User  $user
 * @return bool
 */
public function create(User $user)
{
    //
}
```

Guest Users

By default, all gates and policies automatically return `false` if the incoming HTTP request was not initiated by an authenticated user. However, you may allow these authorization checks to pass through to your gates and policies by declaring an "optional" type-hint or supplying a `null` default value for the user argument definition:

```
<?php

namespace App\Policies;

use App\Post;
use App\User;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\User  $user
     * @param  \App\Post  $post
     * @return bool
     */
    public function update(?User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

Policy Filters

For certain users, you may wish to authorize all actions within a given policy. To accomplish this, define a `before` method on the policy. The `before` method will be executed before any other methods on the policy, giving you an opportunity to authorize the action before the intended policy method is actually called. This feature is most commonly used for authorizing application administrators to perform any action:

```
public function before($user, $ability)
{
    if ($user->isSuperAdmin()) {
        return true;
    }
}
```

If you would like to deny all authorizations for a user you should return `false` from the `before` method. If `null` is returned, the authorization will fall through to the policy method.

> ! The `before` method of a policy class will not be called if the class doesn't contain a method with a name matching the name of the ability being checked.

Authorizing Actions Using Policies

Via The User Model

The `User` model that is included with your Laravel application includes two helpful methods for authorizing actions: `can` and `cant`. The `can` method receives the action you wish to authorize and the relevant model. For example, let's determine if a user is authorized to update a given `Post` model:

```
if ($user->can('update', $post)) {
    //
}
```

If a policy is registered for the given model, the `can` method will automatically call the appropriate policy and return the boolean result. If no policy is registered for the model, the `can` method will attempt to call the Closure based Gate matching the given action name.

Actions That Don't Require Models

Remember, some actions like `create` may not require a model instance. In these situations, you may pass a class name to the `can` method. The class name will be used to determine which policy to use when authorizing the action:

```
use App\Post;

if ($user->can('create', Post::class)) {
    // Executes the "create" method on the relevant policy...
}
```

Via Middleware

Laravel includes a middleware that can authorize actions before the incoming request even reaches your routes or controllers. By default, the `Illuminate\Auth\Middleware\Authorize` middleware is assigned the `can` key in your `App\Http\Kernel` class. Let's explore an example of using the `can` middleware to authorize that a user can update a blog post:

```
use App\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

In this example, we're passing the `can` middleware two arguments. The first is the name of the action we wish to authorize and the second is the route parameter we wish to pass to the policy method. In this case, since we are using implicit model binding, a `Post` model will be passed to the policy method. If the user is not authorized to perform the given action, a HTTP response with a `403` status code will be generated by the middleware.

Actions That Don't Require Models

Again, some actions like `create` may not require a model instance. In these situations, you may pass a class name to the middleware. The class name will be used to determine which policy to use when authorizing the action:

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Post');
```

Via Controller Helpers

In addition to helpful methods provided to the `User` model, Laravel provides a helpful `authorize` method to any of your controllers which extend the `App\Http\Controllers\Controller` base class. Like the `can` method, this method accepts the name of the action you wish to authorize and the relevant model. If the action is not authorized, the `authorize` method will throw an `Illuminate\Auth\Access\AuthorizationException`, which the default Laravel exception handler will convert to an HTTP response with a `403` status code:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
```

```
     * Update the given blog post.
     *
     * @param  Request  $request
     * @param  Post  $post
     * @return Response
     * @throws \Illuminate\Auth\Access\AuthorizationException
     */
    public function update(Request $request, Post $post)
    {
        $this->authorize('update', $post);

        // The current user can update the blog post...
    }
}
```

Actions That Don't Require Models

As previously discussed, some actions like `create` may not require a model instance. In these situations, you should pass a class name to the `authorize` method. The class name will be used to determine which policy to use when authorizing the action:

```
/**
 * Create a new blog post.
 *
 * @param  Request  $request
 * @return Response
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function create(Request $request)
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...
}
```

Authorizing Resource Controllers

If you are utilizing resource controllers, you may make use of the `authorizeResource` method in the controller's constructor. This method will attach the appropriate `can` middleware definitions to the resource controller's methods.

The `authorizeResource` method accepts the model's class name as its first argument, and the name of the route / request parameter that will contain the model's ID as its second argument:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    public function __construct()
    {
        $this->authorizeResource(Post::class, 'post');
    }
}
```

The following controller methods will be mapped to their corresponding policy method:

Controller Method	Policy Method
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete

You may use the `make:policy` command with the `--model` option to quickly generate a policy class for a given model: `php artisan make:policy PostPolicy --model=Post`.

Via Blade Templates

When writing Blade templates, you may wish to display a portion of the page only if the user is authorized to perform a given action. For example, you may wish to show an update form for a blog post only if the user can actually update the post. In this situation, you may use the `@can` and `@cannot` family of directives:

```
@can('update', $post)
    <!-- The Current User Can Update The Post -->
@elsecan('create', App\Post::class)
    <!-- The Current User Can Create New Post -->
@endcan

@cannot('update', $post)
    <!-- The Current User Can't Update The Post -->
@elsecannot('create', App\Post::class)
    <!-- The Current User Can't Create New Post -->
@endcannot
```

These directives are convenient shortcuts for writing `@if` and `@unless` statements. The `@can` and `@cannot` statements above respectively translate to the following statements:

```
@if (Auth::user()->can('update', $post))
    <!-- The Current User Can Update The Post -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- The Current User Can't Update The Post -->
@endunless
```

You may also determine if a user has any authorization ability from a given list of abilities. To accomplish this, use the `@canany` directive:

```
@canany(['update', 'view', 'delete'], $post)
    // The current user can update, view, or delete the post
@elsecanany(['create'], \App\Post::class)
    // The current user can create a post
@endcanany
```

Actions That Don't Require Models

Like most of the other authorization methods, you may pass a class name to the `@can` and `@cannot` directives if the action does not require a model instance:

```
@can('create', App\Post::class)
    <!-- The Current User Can Create Posts -->
@endcan

@cannot('create', App\Post::class)
    <!-- The Current User Can't Create Posts -->
@endcannot
```

Supplying Additional Context

When authorizing actions using policies, you may pass an array as the second argument to the various authorization functions and helpers. The first element in the array will be used to determine which policy should be invoked, while the rest of the array elements are passed as parameters to the policy method and can be used for additional context when making authorization decisions. For example, consider the following `PostPolicy` method definition which contains an additional `$category` parameter:

```
/**
 * Determine if the given post can be updated by the user.
 *
 * @param  \App\User  $user
 * @param  \App\Post  $post
 * @param  int  $category
 * @return bool
 */
public function update(User $user, Post $post, int $category)
{
    return $user->id === $post->user_id &&
           $category > 3;
}
```

When attempting to determine if the authenticated user can update a given post, we can invoke this policy method like so:

```
/**
 * Update the given blog post.
 *
 * @param  Request  $request
 * @param  Post  $post
 * @return Response
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function update(Request $request, Post $post)
{
    $this->authorize('update', [$post, $request->input('category')]);

    // The current user can update the blog post...
}
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.