# Authorization



## # Introduction

In addition to providing authentication services out of the box, Laravel also provides a simple way to organize authorization logic and control access to resources. There are a variety of methods and helpers to assist you in organizing your authorization logic, and we'll cover each of them in this document.

## # Defining Abilities

The simplest way to determine if a user may perform a given action is to define an "ability" using the `Illuminate\Auth\Access\Gate` class. The `AuthServiceProvider` which ships with Laravel serves as a convenient location to define all of the abilities for your application. For example, let's define an `update-post` ability which receives the current `User` and a `Post` model. Within our ability, we will determine if the user's `id` matches the post's `user_id`:

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @param  \Illuminate\Contracts\Auth\Access\Gate  $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);

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

Note that we did not check if the given `$user` is not `NULL`. The `Gate` will automatically return `false` for **all abilities** when there is not an authenticated user or a specific user has not been specified using the `forUser` method.

### Class Based Abilities

In addition to registering `Closures` as authorization callbacks, you may register class methods by passing a string containing the class name and the method. When needed, the class will be resolved via the service container:

```
$gate->define('update-post', 'Class@method');
```

### Intercepting Authorization Checks

Sometimes, you may wish to grant all abilities to a specific user. For this situation, 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 every authorization check. However, you may not modify the result of the authorization check from an `after` callback:

```
$gate->after(function ($user, $ability, $result, $arguments) {
    //
});
```

# # Checking Abilities

## Via The Gate Facade

Once an ability has been defined, we may "check" it in a variety of ways. First, we may use the `check`, `allows`, or `denies` methods on the `Gate` facade. All of these methods receive the name of the ability and the arguments that should be passed to the ability's callback. You do **not** need to pass the current user to these methods, since the `Gate` will automatically prepend the current user to the arguments passed to the callback. So, when checking the `update-post` ability we defined earlier, we only need to pass a `Post` instance to the `denies` method:

```
<?php

namespace App\Http\Controllers;
```

```
use Gate;
use App\User;
use App\Post;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        if (Gate::denies('update-post', $post)) {
            abort(403);
        }

        // Update Post...
    }
}
```

Of course, the `allows` method is simply the inverse of the `denies` method, and returns `true` if the action is authorized. The `check` method is an alias of the `allows` method.

### Checking Abilities For Specific Users

If you would like to use the `Gate` facade to check if a user **other than the currently authenticated user** has a given ability, you may use the `forUser` method:

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

### Passing Multiple Arguments

Of course, ability callbacks may receive multiple arguments:

```
Gate::define('delete-comment', function ($user, $post, $comment) {
    //
});
```

If your ability needs multiple arguments, simply pass an array of arguments to the `Gate` methods:

```
if (Gate::allows('delete-comment', [$post, $comment])) {
    //
}
```

## Via The User Model

Alternatively, you may check abilities via the `User` model instance. By default, Laravel's `App\User` model uses an `Authorizable` trait which provides two methods: `can` and `cannot`. These methods may be used similarly to the `allows` and `denies` methods present on the `Gate` facade. So, using our previous example, we may modify our code like so:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        $post = Post::findOrFail($id);

        if ($request->user()->cannot('update-post', $post)) {
            abort(403);
```

```
        }

        // Update Post...
    }
}
```

Of course, the `can` method is simply the inverse of the `cannot` method:

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

## Within Blade Templates

For convenience, Laravel provides the `@can` Blade directive to quickly check if the currently authenticated user has a given ability. For example:

```
<a href="/post/{{ $post->id }}">View Post</a>

@can('update-post', $post)
    <a href="/post/{{ $post->id }}/edit">Edit Post</a>
@endcan
```

You may also combine the `@can` directive with `@else` directive:

```
@can('update-post', $post)
    <!-- The Current User Can Update The Post -->
@else
    <!-- The Current User Can't Update The Post -->
@endcan
```

## Within Form Requests

You may also choose to utilize your `Gate` defined abilities from a form request's `authorize` method. For example:

```
/**
```

```
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        $postId = $this->route('post');

        return Gate::allows('update', Post::findOrFail($postId));
    }
```

# Policies

## Creating Policies

Since defining all of your authorization logic in the `AuthServiceProvider` could become cumbersome in large applications, Laravel allows you to split your authorization logic into "Policy" classes. Policies are plain PHP classes that group authorization logic based on the resource they authorize.

First, let's generate a policy to manage authorization for our `Post` model. You may generate a policy using the `make:policy` artisan command. The generated policy will be placed in the `app/Policies` directory:

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

### Registering Policies

Once the policy exists, we need to register it with the `Gate` class. The `AuthServiceProvider` contains a `policies` property which maps various entities to the policies that manage them. So, we will specify that the `Post` model's policy is the `PostPolicy` class:

```
<?php

namespace App\Providers;

use App\Post;
use App\Policies\PostPolicy;
```

```php
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

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

    /**
     * Register any application authentication / authorization services.
     *
     * @param  \Illuminate\Contracts\Auth\Access\Gate  $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);
    }
}
```

## Writing Policies

Once the policy has been generated and registered, we can add methods for each ability it authorizes. For example, let's define an `update` method on our `PostPolicy`, which will determine if the given `User` can "update" a `Post`:

```php
<?php

namespace App\Policies;

use App\User;
use App\Post;

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 abilities it authorizes. For example, you might define `show`, `destroy`, or `addComment` methods to authorize various `Post` actions.

> **Note:** All policies are resolved via the Laravel service container, meaning you may type-hint any needed dependencies in the policy's constructor and they will be automatically injected.

### Intercepting All Checks

Sometimes, you may wish to grant all abilities to a specific user on a policy. For this situation, define a `before` method on the policy. This method will be run before all other authorization checks on the policy:

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

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

## Checking Policies

Policy methods are called in exactly the same way as `Closure` based authorization callbacks. You may use the `Gate` facade, the `User` model, the `@can` Blade directive, or the `policy` helper.

### Via The Gate Facade

The `Gate` will automatically determine which policy to use by examining the class of the arguments passed to its methods. So, if we pass a `Post` instance to the `denies` method, the `Gate` will utilize the corresponding `PostPolicy` to authorize actions:

```
<?php

namespace App\Http\Controllers;

use Gate;
use App\User;
use App\Post;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        if (Gate::denies('update', $post)) {
            abort(403);
        }

        // Update Post...
    }
}
```

### Via The User Model

The `User` model's `can` and `cannot` methods will also automatically utilize policies when they are available for the given arguments. These methods provide a convenient way to authorize actions for any `User` instance retrieved by your application:

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

```
    //
}

if ($user->cannot('update', $post)) {
    //
}
```

### Within Blade Templates

Likewise, the `@can` Blade directive will utilize policies when they are available for the given arguments:

```
@can('update', $post)
    <!-- The Current User Can Update The Post -->
@endcan
```

### Via The Policy Helper

The global `policy` helper function may be used to retrieve the `Policy` class for a given class instance. For example, we may pass a `Post` instance to the `policy` helper to get an instance of our corresponding `PostPolicy` class:

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

# Controller Authorization

By default, the base `App\Http\Controllers\Controller` class included with Laravel uses the `AuthorizesRequests` trait. This trait provides the `authorize` method, which may be used to quickly authorize a given action and throw a `HttpException` if the action is not authorized.

The `authorize` method shares the same signature as the various other authorization methods such as `Gate::allows` and `$user->can()`. So, let's use the `authorize` method to quickly authorize a request to update a `Post`:

```
<?php
```

```php
namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        $this->authorize('update', $post);

        // Update Post...
    }
}
```

If the action is authorized, the controller will continue executing normally; however, if the `authorize` method determines that the action is not authorized, a `HttpException` will automatically be thrown which generates a HTTP response with a `403 Not Authorized` status code. As you can see, the `authorize` method is a convenient, fast way to authorize an action or throw an exception with a single line of code.

The `AuthorizesRequests` trait also provides the `authorizeForUser` method to authorize an action on a user that is not the currently authenticated user:

```php
$this->authorizeForUser($user, 'update', $post);
```

### Automatically Determining Policy Methods

Frequently, a policy's methods will correspond to the methods on a controller. For example, in the `update` method above, the controller method and the policy method share the same name: `update`.

For this reason, Laravel allows you to simply pass the instance arguments to the `authorize` method,

and the ability being authorized will automatically be determined based on the name of the calling function. In this example, since `authorize` is called from the controller's `update` method, the `update` method will also be called on the `PostPolicy`:

```
/**
 * Update the given post.
 *
 * @param  int  $id
 * @return Response
 */
public function update($id)
{
    $post = Post::findOrFail($id);

    $this->authorize($post);

    // Update Post...
}
```