Controllers

Introduction

Instead of defining all of your request handling logic as Closures in route files, you may wish to organize this behavior using Controller classes. Controllers can group related request handling logic into a single class. Controllers are stored in the `app/Http/Controllers` directory.

Basic Controllers

Defining Controllers

Below is an example of a basic controller class. Note that the controller extends the base controller class included with Laravel. The base class provides a few convenience methods such as the `middleware` method, which may be used to attach middleware to controller actions:

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return View
     */
    public function show($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

You can define a route to this controller action like so:

```
Route::get('user/{id}', 'UserController@show');
```

Now, when a request matches the specified route URI, the `show` method on the `UserController` class will be executed. The route parameters will also be passed to the method.

> Controllers are not **required** to extend a base class. However, you will not have access to convenience features such as the `middleware`, `validate`, and `dispatch` methods.

Controllers & Namespaces

It is very important to note that we did not need to specify the full controller namespace when defining the controller route. Since the `RouteServiceProvider` loads your route files within a route group that contains the namespace, we only specified the portion of the class name that comes after the `App\Http\Controllers` portion of

the namespace.

If you choose to nest your controllers deeper into the `App\Http\Controllers` directory, use the specific class name relative to the `App\Http\Controllers` root namespace. So, if your full controller class is `App\Http\Controllers\Photos\AdminController`, you should register routes to the controller like so:

```
Route::get('foo', 'Photos\AdminController@method');
```

Single Action Controllers

If you would like to define a controller that only handles a single action, you may place a single `__invoke` method on the controller:

```
<?php

namespace App\Http\Controllers;

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

class ShowProfile extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return View
     */
    public function __invoke($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

When registering routes for single action controllers, you do not need to specify a method:

```
Route::get('user/{id}', 'ShowProfile');
```

You may generate an invokable controller by using the `--invokable` option of the `make:controller` Artisan command:

```
php artisan make:controller ShowProfile --invokable
```

Controller Middleware

Middleware may be assigned to the controller's routes in your route files:

```
Route::get('profile', 'UserController@show')->middleware('auth');
```

However, it is more convenient to specify middleware within your controller's constructor. Using the `middleware` method from your controller's constructor, you may easily assign middleware to the controller's action. You may even restrict the middleware to only certain methods on the controller class:

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log')->only('index');

        $this->middleware('subscribed')->except('store');
    }
}
```

Controllers also allow you to register middleware using a Closure. This provides a convenient way to define a middleware for a single controller without defining an entire middleware class:

```
$this->middleware(function ($request, $next) {
    // ...

    return $next($request);
});
```

You may assign middleware to a subset of controller actions; however, it may indicate your controller is growing too large. Instead, consider breaking your controller into multiple, smaller controllers.

Resource Controllers

Laravel resource routing assigns the typical "CRUD" routes to a controller with a single line of code. For example, you may wish to create a controller that handles all HTTP requests for "photos" stored by your application. Using the `make:controller` Artisan command, we can quickly create such a controller:

```
php artisan make:controller PhotoController --resource
```

This command will generate a controller at `app/Http/Controllers/PhotoController.php`. The controller will contain a method for each of the available resource operations.

Next, you may register a resourceful route to the controller:

```
Route::resource('photos', 'PhotoController');
```

This single route declaration creates multiple routes to handle a variety of actions on the resource. The generated controller will already have methods stubbed for each of these actions, including notes informing you of the HTTP verbs and URIs they handle.

You may register many resource controllers at once by passing an array to the `resources` method:

```
Route::resources([
    'photos' => 'PhotoController',
    'posts' => 'PostController'
]);
```

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Specifying The Resource Model

If you are using route model binding and would like the resource controller's methods to type-hint a model instance, you may use the `--model` option when generating the controller:

```
php artisan make:controller PhotoController --resource --model=Photo
```

Spoofing Form Methods

Since HTML forms can't make `PUT`, `PATCH`, or `DELETE` requests, you will need to add a hidden `_method` field to spoof these HTTP verbs. The `@method` Blade directive can create this field for you:

```
<form action="/foo/bar" method="POST">
    @method('PUT')
</form>
```

Partial Resource Routes

When declaring a resource route, you may specify a subset of actions the controller should handle instead of the full set of default actions:

```
Route::resource('photos', 'PhotoController')->only([
    'index', 'show'
]);

Route::resource('photos', 'PhotoController')->except([
    'create', 'store', 'update', 'destroy'
]);
```

API Resource Routes

When declaring resource routes that will be consumed by APIs, you will commonly want to exclude routes that present HTML templates such as `create` and `edit`. For convenience, you may use the `apiResource` method to automatically exclude these two routes:

```
Route::apiResource('photos', 'PhotoController');
```

You may register many API resource controllers at once by passing an array to the `apiResources` method:

```
Route::apiResources([
    'photos' => 'PhotoController',
    'posts' => 'PostController'
]);
```

To quickly generate an API resource controller that does not include the `create` or `edit` methods, use the `--api` switch when executing the `make:controller` command:

```
php artisan make:controller API/PhotoController --api
```

Naming Resource Routes

By default, all resource controller actions have a route name; however, you can override these names by passing a `names` array with your options:

```
Route::resource('photos', 'PhotoController')->names([
    'create' => 'photos.build'
]);
```

Naming Resource Route Parameters

By default, `Route::resource` will create the route parameters for your resource routes based on the "singularized" version of the resource name. You can easily override this on a per resource basis by using the `parameters` method. The array passed into the `parameters` method should be an associative array of resource names and parameter names:

```
Route::resource('users', 'AdminUserController')->parameters([
    'users' => 'admin_user'
]);
```

The example above generates the following URIs for the resource's `show` route:

```
/users/{admin_user}
```

Localizing Resource URIs

By default, `Route::resource` will create resource URIs using English verbs. If you need to localize the `create` and `edit` action verbs, you may use the `Route::resourceVerbs` method. This may be done in the `boot` method of your `AppServiceProvider`:

```
use Illuminate\Support\Facades\Route;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
```

```
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);
}
```

Once the verbs have been customized, a resource route registration such as `Route::resource('fotos', 'PhotoController')` will produce the following URIs:

```
/fotos/crear

/fotos/{foto}/editar
```

Supplementing Resource Controllers

If you need to add additional routes to a resource controller beyond the default set of resource routes, you should define those routes before your call to `Route::resource`; otherwise, the routes defined by the `resource` method may unintentionally take precedence over your supplemental routes:

```
Route::get('photos/popular', 'PhotoController@method');

Route::resource('photos', 'PhotoController');
```

Remember to keep your controllers focused. If you find yourself routinely needing methods outside of the typical set of resource actions, consider splitting your controller into two, smaller controllers.

Dependency Injection & Controllers

Constructor Injection

The Laravel service container is used to resolve all Laravel controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The declared dependencies will automatically be resolved and injected into the controller instance:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
}
```

You may also type-hint any Laravel contract. If the container can resolve it, you can type-hint it. Depending on your application, injecting your dependencies into your controller may provide better testability.

Method Injection

In addition to constructor injection, you may also type-hint dependencies on your controller's methods. A common use-case for method injection is injecting the `Illuminate\Http\Request` instance into your controller methods:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
```

```
{
    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->name;

        //
    }
}
```

If your controller method is also expecting input from a route parameter, list your route arguments after your other dependencies. For example, if your route is defined like so:

```
Route::put('user/{id}', 'UserController@update');
```

You may still type-hint the `Illuminate\Http\Request` and access your `id` parameter by defining your controller method as follows:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the given user.
     *
     * @param  Request  $request
     * @param  string  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Route Caching

> **!** Closure based routes cannot be cached. To use route caching, you must convert any Closure routes to controller classes.

If your application is exclusively using controller based routes, you should take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. In some cases, your route registration may even be up to 100x faster. To generate a route cache, just execute the `route:cache` Artisan command:

```
php artisan route:cache
```

After running this command, your cached routes file will be loaded on every request. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you should only run the `route:cache` command during your project's deployment.

You may use the `route:clear` command to clear the route cache:

```
php artisan route:clear
```

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

Laravel

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

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.

Dusk

Passport

Scout

Socialite