Laravel: Controllers and Middleware

DWES

1.- Introduction

In this unit we will see how to add controllers to Laravel projects.

At the end of that point, the views and controllers will be used, leaving the models for the next unit.

We will also see how to create filters using Middleware applied to routes, redirections and forms in Laravel.

2.- Controllers

So far, we have only seen how to return a string for a route and how to associate a view to a route directly in the routes file. But in general, the recommended way of working will be to associate these routes with a method of a controller. This will allow you to separate the code much better and create classes (controllers) that group together all the functionality of a given resource.

As seen in the mvc unit, controllers are the entry point for user requests and are the ones that must contain all the logic associated with the processing of a request, being responsible for making the necessary queries to the database, preparing the data and calling the corresponding view with said data.

2.- Controllers

In Laravel the controllers are stored in the app/Http/Controllers directory and although it is not necessary, due to usage conventions the Controller suffix is added to the name, so we will have UserControler.php MoviesController.php...

For the MVC pattern, object-oriented programming is used, so the controllers will be classes and must also inherit from the Controller superclass which is provided by Laravel itself and is in the app/Http/Controllers directory

Thanks to this superclass, all the logic actions of all the controllers created for the web application will be centralized. By default, it only loads code for validation and authorization, but if necessary, you could add custom code that is needed in all your own controllers.

2.- Controllers

The app/Http/Controllers/UserController.php controller code is shown below as an example:

```php
namespace App\Http\Controllers;

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

class UserController extends Controller
{
	/**
	* Mostrar información de un usuario.
	* @param int $id
	* @return Response
	*/
	public function showProfile($id)
	{
		$user = User::findOrFail($id);
		return view('user.profile', ['user' => $user]);
	}
}
```

```php
namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

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

2.- Controllers

In the previous code the showProfile($id) method is responsible for retrieving the user data and loading the user.profile view from that user data.

This controller only has this action. For each action that you want to perform in a controller, you must add a method with the name of the action.

In order to use a controller in the routes, it must be done as follows:

```
Route::get('user/{id}', [MY_CLASS::class, 'FUNCTION_IN_CLASS']);
```

So, we can create the following route:

```
Route::get('user/{id}', [UserController::class, 'showProfile']);
```

When creating a route to a class, remember to add 'use $PATH\MY_CLASS;' at the beginning of the file.

2.- Controllers

Controllers can be created by hand or using the following Artisan command to create an empty file and then proceed to enter code into it:

```
php artisan make:controller CONTROLLER_NAME
```

If you are going to create a handler to do CRUD (Create-Read-Update-Delete) on the data model, you can use the --resource option:

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

In this way, the controller that is created will have the necessary methods for the CRUD system: index, store, create, show, update, destroy and edit. These methods will be empty to be able to enter the necessary code.

2.- Controllers

To make use of this CRUD, you can add the controller in the routes file as a resource as follows:

```
Route::resource('URI', CONTROLLER_NAME::class);
```

You can see the routes created:

```
C:\xampp\htdocs\laravel\pruebadwes>php artisan route:list
+--------+--------------+--------------------+-------------+-----------------------------------------------------------+-----------------------------------------+
| Domain | Method       | URI                | Name        | Action                                                    | Middleware                              |
+--------+--------------+--------------------+-------------+-----------------------------------------------------------+-----------------------------------------+
|        | GET|HEAD     | /                  |             | Closure                                                   | web                                     |
|        | GET|HEAD     | URI                | URI.index   | App\Http\Controllers\CONTROLLER_NAME@index                | web                                     |
|        | POST         | URI                | URI.store   | App\Http\Controllers\CONTROLLER_NAME@store                | web                                     |
|        | GET|HEAD     | URI/create         | URI.create  | App\Http\Controllers\CONTROLLER_NAME@create               | web                                     |
|        | GET|HEAD     | URI/{URI}          | URI.show    | App\Http\Controllers\CONTROLLER_NAME@show                 | web                                     |
|        | PUT|PATCH    | URI/{URI}          | URI.update  | App\Http\Controllers\CONTROLLER_NAME@update               | web                                     |
|        | DELETE       | URI/{URI}          | URI.destroy | App\Http\Controllers\CONTROLLER_NAME@destroy              | web                                     |
|        | GET|HEAD     | URI/{URI}/edit     | URI.edit    | App\Http\Controllers\CONTROLLER_NAME@edit                 | web                                     |
|        | GET|HEAD     | api/user           |             | Closure                                                   | api                                     |
|        |              |                    |             |                                                           | App\Http\Middleware\Authenticate:sanctum |
|        | GET|HEAD     | child              |             | Closure                                                   | web                                     |
|        | GET|POST|HEAD | greeting           |             | Closure                                                   | web                                     |
|        | GET|HEAD     | sanctum/csrf-cookie |            | Laravel\Sanctum\Http\Controllers\CsrfCookieController@show | web                                     |
|        | GET|HEAD     | table/{number}     |             | Closure                                                   | web                                     |
|        | GET|HEAD     | user/{id}          |             | App\Http\Controllers\UserController@showProfile           | web                                     |
+--------+--------------+--------------------+-------------+-----------------------------------------------------------+-----------------------------------------+
```

2.- Controllers

With the helper action we can generate a url for the given controller action:

```php
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);
```

If the method accepts route parameters, you may pass them as the second argument to the method:

```php
$url = action([UserController::class, 'profile'], ['id' => 1]);
```

3.- Middleware

Middleware provide a convenient mechanism for inspecting and filtering HTTP requests entering your application: https://laravel.com/docs/12.x/middleware

A filter or middleware is defined as a PHP class stored in a file within the app/Http/Middleware folder. Each middleware will be in charge of applying a specific type of filter and deciding what to do with the request made: allow its execution, give an error or redirect to another page if it is not allowed.

Laravel includes middleware to manage authentication, maintenance mode, CSRF protection, and a few more. We can find all these filters in the app/Http/Middleware folder, they can be modified. But in addition to these you can create your own Middleware as will be seen later.

3.- Middleware

To create a new middleware, use the make:middleware Artisan command:

```
php artisan make:middleware MIDDLEWARE_NAME
```

We are going to create the following middleware:

```
php artisan make:middleware EnsureTokenIsValid
```

This command will place a new EnsureTokenIsValid class within your app/Http/Middleware directory and is prepared to enter the filter code in the handle function. This function has two parameters:

- $request: includes all the input parameters of the request
- $next: callback to continue execution

3.- Middleware

In this middleware, we will only allow access to the route if the supplied token input matches a specified value. Otherwise, we will redirect the users back to the home URI (we will see redirect later)

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFounda
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

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

3.- Middleware

This middleware, will act when the user is under 18 years old, redirecting the request to the main page in that case or continuing the normal course of the request if the user is of legal Age.

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure(\Illuminate\Http\Request): (\Illumina
     * @return \Illuminate\Http\Response|\Illuminate\Http\Red
     */
    public function handle(Request $request, Closure $next)
    {
        if ($request->input('age') < 18) {
            return redirect('home');
            }

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

3.- Middleware

Three different actions can be performed in a middleware:

- Continue the request when everything is right:

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

- Redirect the request to prevent access if the filter is not passed:

```
return redirect('home’);
```

- Throw an exception with the method abort

```
abort (403, 'Unauthorized action.');
```

3.- Middleware

A middleware can perform tasks before or after passing the request deeper into the application. For example, the following middleware would perform some task before the request is handled by the application:

```php
class BeforeMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        // Perform action

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

3.- Middleware

However, this middleware would perform its task after the request is handled by the application:

```php
class AfterMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        $response = $next($request);

        // Perform action

        return $response;
    }
}
```

3.- Middleware

We have seen the function and creation of a middleware, now we are going to see the usage, registering middleware.

- Global Middleware

If you want a middleware to run during every HTTP request to your application, you may append it to the global middleware stack in your application's bootstrap/app.php file.

```
use App\Http\Middleware\EnsureTokenIsValid;

->withMiddleware(function (Middleware $middleware): void {
    $middleware->append(EnsureTokenIsValid::class);
})
```

3.- Middleware

- Assigning Middleware to Routes

If you would like to assign middleware to specific routes, you may invoke the middleware method when defining the route:

```php
use App\Http\Middleware\EnsureTokenIsValid;

Route::get('/profile', function () {
    // ...
})->middleware(EnsureTokenIsValid::class);
```

You may assign multiple middleware to the route by passing an array of middleware names to the middleware method:

```php
Route::get('/', function () {
    // ...
})->middleware([First::class, Second::class]);
```

3.- Middleware

- Middleware groups

Sometimes you may want to group several middleware under a single key to make them easier to assign to routes. You may accomplish this using the appendToGroup method within your application's bootstrap/app.php file:

```php
use App\Http\Middleware\First;
use App\Http\Middleware\Second;

->withMiddleware(function (Middleware $middleware): void {
    $middleware->appendToGroup('group-name', [
        First::class,
        Second::class,
    ]);

    $middleware->prependToGroup('group-name', [
        First::class,
        Second::class,
    ]);
})
```

3.- Middleware

- Middleware groups

Middleware groups may be assigned to routes and controller actions using the same syntax as individual middleware. Again, middleware groups make it more convenient to assign many middleware to a route at once:

```
Route::get('/', function () {
    // ...
})->middleware('web');

Route::middleware(['web'])->group(function () {
    // ...
});
```

4.- Advanced Routing

Redirect routes: https://laravel.com/docs/12.x/routing#redirect-routes

A redirect can also be returned in response to a request:

```
Route::redirect('/here', '/there');
```

Redirect method: https://laravel.com/docs/12.x/helpers#method-redirect

If needed, we can make a redirect to a route or a named route at any time:

```
return redirect('/home');


return redirect()->route('route.name');
```

4.- Advanced Routing

We can create routes that redirect:

```
Route::get('/dashboard', function () {
    return redirect('home/dashboard');
});
```

Sometimes you may wish to redirect the user to their previous location, such as when a submitted form is invalid. You may do so by using the global back helper function:

```
1    return back($status = 302, $headers = [], $fallback = '/');
2
3    return back();
```

4.- Advanced Routing

Redirecting with input

You may use the withInput method to flash the current request's input data to the session before redirecting the user to a new location.

```
return back()->withInput();
```

The old method will pull the previously flashed input data from the session:

```
$username = $request->old('username');
```

```
<input type="text" name="username" value="{{ old('username') }}">
```

4.- Advanced Routing

You may also generate redirects to controller actions. To do so, pass the controller and action name to the action method:

```php
use App\Http\Controllers\HomeController;

return redirect()->action([HomeController::class, 'index']);
```

If your controller route requires parameters, you may pass them as the second argument to the action method:

```php
return redirect()->action(
    [UserController::class, 'profile'], ['id' => 1]
);
```

4.- Advanced Routing

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the name method onto the route definition:

```
Route::get('/user/profile', function () {
    // ...
})->name('profile');
```

You may also specify route names for controller actions:

```
Route::get(
    '/user/profile',
    [UserProfileController::class, 'show']
)->name('profile');
```

4.- Advanced Routing

Route groups allow you to share route attributes, such as middleware, across a large number of routes without needing to define those attributes on each individual route.

Nested groups attempt to intelligently "merge" attributes with their parent group. Middleware and where conditions are merged while names and prefixes are appended. Namespace delimiters and slashes in URI prefixes are automatically added where appropriate.

4.- Advanced Routing

Middleware

To assign middleware to all routes within a group, you may use the middleware method before defining the group. Middleware are executed in the order they are listed in the array:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second middleware...
    });

    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

4.- Advanced Routing

Controllers

If a group of routes all utilize the same controller, you may use the controller method to define the common controller for all of the routes within the group. Then, when defining the routes, you only need to provide the controller method that they invoke:

```php
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

4.- Advanced Routing

Subdomain Routing

Route groups may also be used to handle subdomain routing. Subdomains may be assigned route parameters just like route URIs, allowing you to capture a portion of the subdomain for usage in your route or controller. The subdomain may be specified by calling the domain method before defining the group:

```
Route::domain('{account}.example.com')->group(function () {
    Route::get('user/{id}', function (string $account, string $id) {
        // ...
    });
});
```

4.- Advanced Routing

Route name prefixes

The name method may be used to prefix each route name in the group with a given string. For example, you may want to prefix the names of all of the routes in the group with admin. The given string is prefixed to the route name exactly as it is specified, so we will be sure to provide the trailing . character in the prefix:

```
Route::name('admin.')->group(function () {
    Route::get('/users', function () {
        // Route assigned name "admin.users"...
    })->name('users');
});
```

4.- Advanced Routing

Route helper method: https://laravel.com/docs/12.x/helpers#method-route

The route function generates a URL for a given named route:

```php
$url = route('route.name');
```

If the route accepts parameters, you may pass them as the second argument to the function:

```php
$url = route('route.name', ['id' => 1]);
```

You can use this url anywhere in the application.

5.- Forms

The creation of forms is also HTML code, so at this point we will simply explain a few brief notions of how to work with them in Laravel.

Two attributes must generally be indicated in the HTML form tag:

- Method: how the data will be sent (HTTP request type).
- Action: which resource will receive the form data.

```
<form method="POST" action="">
    ...
</form>
```

5.- Forms

Since Laravel uses Blade, you can configure the action attribute with the URLs that Laravel creates, for example:

```
<form action="{{ url('foo/bar') }}" method="POST">
    ...
</form>
```

Or send the data to a method of a controller directly:

```
<form action="{{ action([HomeController::class,'getHome']) }}“ method="POST">
...
</form>
```

5.- Forms

Anytime you define an HTML form in your application, you should include a hidden CSRF (Cross-Site Request Forgery) token field in the form so that the CSRF protection middleware can validate the request. You may use the @csrf Blade directive to generate the token field:

```
<form method="POST" action="/profile">
    @csrf

    ...
</form>
```

5.- Forms

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>
```

5.- Forms

Form fields

As in a traditional web application, you may need to display the previously entered value in the form fields if a failure has occurred in the form or fill out a form with the existing values in the database if you are going to edit it.

<input type="text" name="nombre" id="nombre" value="**{{ $nombre }}**">

If the withInput method explained previously has been used, a form field can be filled with the value that was sent to the request thanks to the old function:

<input type="text" name="nombre" id="nombre" value="**{{ old('nombre') }}**">

Exercise

We are going to continue with the web application for managing a video store.

Initially the controllers and methods associated with each route will be added and finally the views will be completed with forms using Blade.

Exercise

Controllers

First, we will have to create the two controllers necessary for the web application, CatalogController.php and HomeController.php. Once created, methods will be added according to the following table:

Route	Controller	Method
/	HomeController	getHome
catalog	CatalogController	getIndex
catalog/show/{id}	CatalogController	getShow
catalog/create	CatalogController	getCreate
catalog/edit/{id}	CatalogController	getEdit

Exercise

Routes

Modify the corresponding routes so that they point to the methods of the created controllers.

The code that the old routes returned (return) will have to be moved to the corresponding controller.

Exercise

Complete the views
The code of the methods of the created controllers must be completed.

[HomeController::class,'getHome’]

At the moment, this method will only redirect you to the movie catalog, later it will check if the user is logged in and if not, you will be redirected to the login form.

Exercise

[CatalogController::class,'getIndex’]

This view should show the video store's entire movie catalog. At the moment the database will not be accessed, so to obtain the movies the file array_peliculas.php will be used, you must copy it to the same directory as the CatalogController.php file and include it in the CatalogController class using the require function.

In the getIndex method you must pass the entire array of movies ($this->arrayPeliculas)

Exercise

In the corresponding view, the following code must be included in the content section:

```
<div class="row">
    @foreach( $arrayPeliculas as $key => $pelicula )
        <div class="col-xs-6 col-sm-4 col-md-3 text-center">
            <a href="{{ url('/catalog/show/' . $key ) }}">
                <img src="{{$pelicula['poster']}}" style="height:200px"/>
                <span style="min-height:45px;margin:5px 0 10px 0">
                {{$pelicula['title']}}
                </span>
            </a>
        </div>
    @endforeach
</div>
```

Exercise

[CatalogController::class, getShow’]
This method receives the identifier of the movie whose data should be displayed. Since an array is being used, the id will be the position of that movie in the array ($this->arrayPeliculas[$id])

Create two columns, one with the image of the movie and another with its details.

```
<div class="row">
    <div class="col-sm-4">
        {{-- TODO: Imagen de la película --}}
    </div>
    <div class="col-sm-8">
        {{-- TODO: Datos de la película --}}
    </div>
</div>
```

Exercise

Below is a screenshot of how this application screen should look like:

Exercise

The status of the movie is found in the rented value of the movie array. If it is false, the text “Movie available” and a blue button for “Rent movie” will be displayed. If true, the text “Movie currently rented” will appear. and the red button for “Return Movie” will be displayed. These two buttons do not have to work yet.

Two more buttons must be added: “Edit movie” and “Return to list” that must direct to the correct paths. Buttons should be button-like links thanks to Bootstrap: https://getbootstrap.com/docs/4.2/components/buttons/

Exercise

[CatalogController::class, ‘getCreate’]
You must return the catalog.create view to add a movie. The content of this method must be copied from the catalog_create.php file by completing the indicated parts with ALL and adding an “Add movie” button at the end of the form. The form will not work, it will be filled out later.

[CatalogController::class, ‘getEdit’]
It will show a form, which can be copied and pasted from the previous point, this form must show the data of the movie whose identifier is received ($this->arrayPeliculas[$id]). The following points must be changed:

- Title: “Modificar película”.
- Button Text: “Modificar película”
- Add the request method PUT