Unit 5. Controllers

1. Controllers

The controllers allow us to better structure the code of our application. We use it to release the route files from also having to deal with some common logic of the requests, such as access to data, validation of forms, etc. In addition, as the application grows, the route file may become too large if you have to also store the logic of each route, and the processing time of the file will also grow. It's best to break that logic down into controllers.

To define a controller in our application, we have to use again the command 'php artisan' previously seen. Specifically, we will use the option 'make:controller' followed by the name we want to give to the controller. Typically, controller names end with the *Controller* suffix,so we can create a test one like this:

```
php artisan make:controller TestController
```

This will generate an empty class with the name of the controller. By default, controllers are saved in the 'app/Http/Controllers' subfolder of our Laravel project.

1.1. Single-method drivers (*invoke*)

The above command supports a few more additional parameters. A very useful one is the '-i' parameter, which creates the controller with a method called '__invoke',which self-executes when called from some routing process. For example, if we create the controller like this:

```
php artisan make:controller TestController -i
```

The 'TestController' class will be created in the 'app/Http/Controllers' folder, with content like this:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TestController extends Controller
{
    ...
    public function __invoke(Request $request)
    {
        ...
```

```
    }
}
```

Within the '__invoke' method we can define the logic of generating or obtaining the data that a view needs, and render it. For example:

```
    public function __invoke(Request $request)
    {
        $data =  array(...);
        return view('miVista', compact('data'));
    }
```

Thus, in the route file, it is enough to define the route that we want, and as a second parameter of the 'get' method, indicate the class of the controller that is going to be triggered to process that route. Additionally, we can also assign a name to the route, as we have already done in previous examples.

You must include with 'use' the Controller class in the route file:

```
use App\Http\Controllers\TestController;
...
Route::get('test', TestController::class)->name('test');
```

1.2. Multi-method controllers

1.2.1. Resource Controllers

If we create a controller with the '-r' option instead of the '-i' option used in the previous example, it will create a resource controller('resources'),and predefine in it a series of utility methods for the main operations that can be performed on an entity of our application:

- 'index': shows a list of the elements of that entity or resource
- 'create': shows the form to register new elements
- 'store': stores in the database the resource created with the previous form
- 'show': displays the data of a specific resource (from its key or *id*).
- 'edit': displays the form to edit an existing resource
- 'update': updates in the database the resource edited with the previous form
- 'destroy': deletes a resource by its identifier.

Obviously, the code of all these methods will appear empty at the beginning, and we will have to fill them with the corresponding operations later.

If we want to use a controller of this type, and call any of its methods from some route, it is no longer enough to put the name of the controller, as we did before with those of type *invoke*,since now there is more than one method to choose. In this case, we define an array with the class of the controller and the method to be called('index' in our case):

For example:

```
use App\Http\Controllers\TestController;
...
Route::get('test', [TestController::class, 'index'])
    ->name('list_test');
```

Let's try this option in our library project. We will create a controller to manage the books, with this command:

```
php artisan make:controller -r BookController
```

At the moment several of the methods generated in the controller will not be used. We can modify the two that we are going to use at the moment('index' and 'show')and put in them what we previously had in the route file. This is how we would be, respectively:

```
public function index()
    {
        $books = array(
            array("title" => "Ender's game",
            "author" => "Orson Scott Card"),
            array("title" => "The table of Flanders",
            "author" => "Arturo Pérez Reverte"),
            array("title" => "The Endless Story",
            "author" => "Michael Ende"),
            array("title" => "The Lord of the Rings",
            "author" => "J.R.R. Tolkien")
            );
            return view('books.list', compact('books'));

    }


public function show(string $id)
{
return "Showing book file $id";
}
```

> **NOTE:** the 'show' method we had not implemented in the previous session, but basically we are going to use it to show the information of a book. At the moment we show only the *id* of the received book, as plain text.

Changes in the file 'routes/web.php'. First we have to delete the old code of listing posts because we have this code in Bookcontroller in index function. These two routes would now look like this:

```
use App\Http\Controllers\BookController;
...
Route::get('books', [BookController::class, 'index'])-> name('book_list');
Route::get('books/{id}', [BookController::class, 'show']);
```

1.2.2. API controllers

As an alternative to the resource controllers seen before, we can create The controllers with the '--api' option. It will create a controller with the same methods as the resource method, except for the 'create' and 'edit' methods, responsible for displaying the forms for creating and editing resources, since in APIs these forms are not necessary, as we will see in later sessions.

1.2.3. Renaming the views

As the project grows, we will generate a good number of views associated with controllers, and it is necessary to structure these views in an appropriate way to be able to identify them quickly. A convention we can follow is to name the views from the controller or model they reference, and the operation they perform. For example, if we have a controller called 'TestController',it is supposed to act on a table called 'tests' (we will see it later, in the data access session). In our case of the library, we can store the views of the books of the library in the subfolder 'resources/views/books',and define within the views associated with each operation of the controller that we have defined. So we have to change the name from list.blade.php to index.blade.php and create show.blade.php:

- index.blade.php
- show.blade.php
- ...

In show.blade.php the code will be:

```
@extends('template')
@section('title', 'show')
@section('content')
<h1>Book</h1>
Showing book file {{$id}}
@endsection
```

In parallel, every time we go to load a view from a controller or route, we will refer to this name. Thus, if we want to render the 'show' view for the books from the 'show' method of the book controller, we would do something like this (passing as a parameter the *id* of the book to be searched, so that you can take it out in the view for now):

```
 public function show(string $id)
{
    return view('books.show', compact('id'));
}
```

Similarly, the names we associate to the routes should follow this same pattern.

1.2.4. Joining all the paths of a controller

At the end of the whole process of implementing a controller (resource or API) we will have in the route file one dedicated to each method of the controller (one for 'index',another for 'show',etc.). These routes can be grouped into one using the 'resource' method of the 'Route' class, instead of 'get',indicating as parameters the base name of the route, and the controller that is going to take care of it:

```
use App\Http\Controllers\BookController;
...
Route::resource('books', BookController::class);
```

The above route will define a GET route to '/books',served by the 'index' method of the controller, another GET route to '/books/{id}' served by the 'show' method of the controller... etc.

We can also use the 'only' method to indicate for which methods we want routes:

```
use App\Http\Controllers\BookController;
...
Route::resource('books', BookController::class)
    ->only(['index', 'show']);
```

From the opposite side, we have available the 'except' method to indicate that all routes are generated except those for the indicated methods:

```
use App\Http\Controllers\BookController;
...
Route::resource('books', BookController::class)
    ->except(['update', 'edit']);
```

With API type controllers we can also automatically generate all the routes for their methods, using the 'apiResource' method of the 'Route' class, instead of the 'resource' method used before:

```
use App\Http\Controllers\TestController;
...
Route::apiResource('test',  TestController::class);
```

1.2.5. Create and Edit

For our library example, we can return a plain text in the 'create' and 'edit' methods that indicate that it should be a form:

```
public function create()
{
    return "Book Insertion Form";
}


public function edit(string $id)
{
return "Book Editing Form";
}
```

Our route file can be left with this single instruction for all book routes, indicating that for now we will only manage the list, the file and the two forms (create, edit):

```
Route::resource('books', BookController::class)
->only(['index', 'show', 'create', 'edit']);
```

Now we can access these 4 URLs and see the corresponding answer:

- http://library/books (call 'index')
- http://library/books/3 (will call 'show')
- http://library/books/create (call 'create')
- http://library/books/3/edit (call 'edit')