Sesión 3. Controladores e inyección de dependencias

Los controladores permiten estructurar mejor el código de nuestra aplicación. Su principal utilidad radica en liberar a los archivos de rutas de tener que ocuparse también de gestionar cierta lógica común de las peticiones, como el acceso a los datos, validación de formularios, etc. Además, a medida que la aplicación crezca, el archivo de rutas puede ser demasiado grande si tiene que almacenar también la lógica de cada ruta, y el tiempo de procesamiento del archivo también crecerá. Lo mejor es dividir esa lógica en controladores.

1. Definición de controladores

Para definir un controlador en nuestra aplicación, tenemos que echar mano de nuevo del comando `php artisan` visto previamente. En concreto, utilizaremos la opción `make:controller` seguida del nombre que le queramos dar al controlador. Típicamente, los nombres de controladores terminan con el sufijo *Controller*, por lo que podemos crear uno de prueba así:

```
php artisan make:controller PruebaController
```

Esto generará una clase vacía con el nombre del controlador. Por defecto, los controladores se guardan en la subcarpeta `app/Http/Controllers` de nuestro proyecto Laravel.

1.1. Controladores de un sólo método (*invoke*)

El comando anterior admite algunos parámetros adicionales más. Uno muy útil es el parámetro `-i`, que crea el controlador con un método llamado `__invoke`, que se auto ejecuta cuando es llamado desde algún proceso de enrutamiento. Por ejemplo, si creamos el controlador así:

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

Se creará la clase `PruebaController` en la carpeta `app/Http/Controllers`, con un contenido como éste:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

Dentro del método `__invoke` podemos definir la lógica de generar u obtener los datos que necesita una vista, y renderizarla. Por ejemplo:

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

Así, en el archivo de rutas, basta con definir la ruta que queramos, y como segundo parámetro del método `get`, indicar el nombre del controlador que se va a disparar para procesar esa ruta. Adicionalmente, también le podemos asignar un nombre a la ruta, como ya hemos hecho en ejemplos anteriores.

```php
Route::get('prueba', 'PruebaController')->name('prueba');
```

1.2. Controladores de varios métodos

1.2.1. Controladores de recursos

Si creamos un controlador con la opción `-r` en lugar de la opción `-i` utilizada en el ejemplo anterior, creará un controlador de recursos (`resources`), y predefinirá en él una serie de métodos de utilidad para las operaciones principales que se pueden realizar sobre una entidad de nuestra aplicación:

- `index`: muestra un listado de los elementos de esa entidad o recurso
- `create`: muestra el formulario para dar de alta nuevos elementos
- `store`: almacena en la base de datos el recurso creado con el formulario anterior
- `show`: muestra los datos de un recurso específico (a partir de su clave o *id*).

- `edit` : muestra el formulario para editar un recurso existente
- `update` : actualiza en la base de datos el recurso editado con el formulario anterior
- `destroy` : elimina un recurso por su identificador.

Obviamente, el código de todos estos métodos aparecerá vacío al principio, y los deberemos rellenar con las operaciones correspondientes más adelante.

Si queremos utilizar un controlador de este tipo, y llamar a alguno de sus métodos desde alguna ruta, ya no basta con poner el nombre del controlador, como hacíamos antes con los de tipo *invoke*, puesto que ahora hay más de un método que elegir. Lo que haremos será poner el nombre del controlador, seguido de una arroba `@` y el nombre del método a invocar. Por ejemplo:

```
Route::get('prueba', 'PruebaController@index')->name('listado_prueba');
```

Vamos a probar esta opción en nuestro proyecto de biblioteca. Crearemos un controlador para gestionar los libros, con este comando:

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

De momento varios de los métodos generados en el controlador no los vamos a utilizar. Podemos modificar los dos que sí vamos a usar de momento (`index` y `show`) y poner en ellos lo que antes teníamos en el archivo de rutas. Así nos quedarían, respectivamente:

```
public function index()
{
    $libros = array(
        array("titulo" => "El juego de Ender",
              "autor" => "Orson Scott Card"),
        array("titulo" => "La tabla de Flandes",
              "autor" => "Arturo Pérez Reverte"),
        array("titulo" => "La historia interminable",
              "autor" => "Michael Ende"),
        array("titulo" => "El Señor de los Anillos",
              "autor" => "J.R.R. Tolkien")
    );

    return view('listado', compact('libros'));
}

public function show($id)
{
    return "Mostrando ficha de libro $id";
}
```

> **NOTA:** el método `show` no lo habíamos implementado en la sesión anterior, pero básicamente lo vamos a utilizar para mostrar la ficha de un libro. De momento mostramos sólo el *id* del libro recibido, como texto plano.

Estas dos rutas quedarían ahora así en el archivo `routes/web.php` (eliminaríamos la vieja ruta de listado de posts):

```
Route::get('libros', 'LibroController@index');
Route::get('libros/{id}', 'LibroController@show');
```

1.2.2. Controladores de API

Como alternativa a los controladores de recursos vistos antes, podemos crear los controladores con la opción `--api`. Creará un controlador con los mismos métodos que el de recursos, salvo los métodos `create` y `edit`, encargados de mostrar los formularios de creación y edición de recursos, ya que en las APIs estos formularios no son necesarios, como veremos en sesiones posteriores.

1.2.3. Renombrando las vistas

A medida que el proyecto crece, generaremos un buen número de vistas asociadas a controladores, y es necesario estructurar estas vistas de una forma adecuada para poderlas identificar rápidamente. Una convención que podemos seguir es nombrar las vistas a partir del controlador o modelo al que hacen referencia, y a la operación que realizan. Por ejemplo, si tenemos un controlador llamado `PruebaController`, se supone que actuará sobre una tabla llamada `pruebas` (lo veremos más adelante, en la sesión de acceso a datos). En nuestro caso de la biblioteca, podemos almacenar las vistas de los libros de la biblioteca en la subcarpeta `resources/views/libros`, y definir dentro las vistas asociadas a cada operación del controlador que tengamos definida. Por ejemplo:

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

Paralelamente, cada vez que vayamos a cargar una vista desde algún controlador o ruta, haremos referencia a este nombre. Así, si queremos renderizar la vista `show` para los libros desde el método `show` del controlador de libros, haríamos algo así (pasándole como parámetro el *id* del libro a buscar, para que lo saque en la vista por ahora):

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

Del mismo modo, los nombres que asociemos a las rutas deberían seguir este mismo patrón.

1.2.4. Aunando todas las rutas de un controlador

Al final de todo el proceso de implementación de un controlador (de recursos o de API) tendremos en el archivo de rutas una dedicada a cada método del controlador (una para `index`, otra para `show`, etc.). Estas rutas pueden agruparse en una sola usando el método `resource` de la clase `Route`, en lugar de `get`, indicándole como parámetros el nombre base de la ruta, y el controlador que se va a encargar de ella:

```
Route::resource('libros', 'LibroController');
```

La ruta anterior definirá una ruta GET hacia `/libros`, atendida por el método `index` del controlador, otra ruta GET hacia `/libros/{id}` atendida por el método `show` del controlador... etc.

También podemos utilizar el método `only` para indicar para qué métodos queremos rutas:

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

Desde el lado opuesto, tenemos disponible el método `except` para indicar que se generen todas las rutas excepto aquellas para los métodos indicados:

```
Route::resource('libros', 'LibroController')->except(['update', 'edit']);
```

Con los controladores de tipo API también podemos generar automáticamente todas las rutas para sus métodos, utilizando el método `apiResource` de la clase `Route`, en lugar del método `resource` empleado antes:

```
Route::apiResource('prueba', 'PruebaController');
```

1.2.5. Renombrando las rutas

Si generamos rutas automáticas para los métodos de un controlador, veremos que para los formularios de crear y editar se define una ruta terminada en `/create` o en `/edit`, respectivamente. Esto puede chocar si pretendemos una web hecha en otro idioma. Pero podemos cambiar el nombre que se genera automáticamente para estas rutas, editando el proveedor de servicios `AppServiceProvider`, ubicado en la carpeta `app/Providers`. En el método `boot`, podemos llamar al método `resourceVerbs` de la clase `Route` y renombrar los verbos utilizados para acceder a las rutas del recurso. Por ejemplo:

```
public function boot()
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar'
    ])
}
```

> **NOTA**: deberemos incluir con `use` el espacio de nombres `Illuminate\Support\Facades\Route` para poder emplear la clase `Route` en el proveedor de servicios.

Para nuestro ejemplo de la biblioteca, podemos devolver un texto plano en los métodos `create` y `edit` que indiquen que ahí va un formulario:

```
public function create()
{
    return "Formulario de inserción de libros";
}

public function edit()
{
    return "Formulario de edición de libros";
}
```

Nuestro archivo de rutas se puede quedar con esta única instrucción para todas las rutas de libros, indicando que por ahora sólo vamos a gestionar el listado, la ficha y los dos formularios:

```
Route::resource('libros', 'LibroController')
->only(['index', 'show', 'create', 'edit']);
```

Ahora y podemos acceder a estas 4 URLs y ver la respuesta correspondiente:

- http://biblioteca/libros (llamará a `index`)
- http://biblioteca/libros/3 (llamará a `show`)
- http://biblioteca/libros/crear (llamará a `create`)
- http://biblioteca/libros/3/editar (llamará a `edit`)

Notar que la URL aparece en castellano gracias a los cambios en `AppServiceProvider`, pero los nombres de funciones en los controladores siguen estando en inglés, así como en los elementos de la llamada al método `only`.

Podemos consultar el conjunto de rutas al que está respondiendo nuestra aplicación en todo momento con este comando:

```
php artisan route:list
```

En este punto, puedes realizar el Ejercicio 1 de los propuestos al final de la sesión.

2. Inyección de dependencias en Laravel

El concepto de *inyección de dependencias* es muy habitual en el uso de frameworks. Consiste en un mecanismo que facilita recursos a los diferentes componentes de la aplicación, y es algo que ya hemos utilizado, sin saberlo, en los métodos que se han generado para los controladores.

2.1. Ejemplo: la petición del usuario

Por ejemplo, cuando definimos un método en un controlador que necesita procesar una petición, se le pasa como parámetro un objeto de tipo `Request`. Automáticamente, Laravel procesa el tipo de dato y obtiene el objeto asociado (en este caso, la petición del cliente).

```
class LibroController extends Controller
{
    ...
    public function store(Request $request)
    {
        ...
    }
}
```

2.2. Ejemplo: la respuesta

Al igual que tenemos un objeto `Request` para obtener datos de la petición, también existe un `Response` para gestionar la respuesta. Laravel proporciona un método `response` al que le podemos pasar varios parámetros:

1. El contenido de la respuesta
2. El código de estado HTTP de respuesta (si no se especifica, por defecto es 200)
3. Un array con las cabeceras de respuesta (por defecto está vacío).

Así, si por ejemplo queremos emitir una respuesta determinada con su código de estado desde un controlador, podemos hacer esto (por ejemplo, para un código 201):

```
response("Mensaje de respuesta", 201);
```

Las cabeceras pueden especificarse como un array, o enlazando llamadas al método `header` (una para cada cabecera):

```
response("Mensaje de respuesta", 201)
    ->header('Cabecera1', 'Valor1')
    ->header('Cabecera2', 'Valor2');
```

En el caso de querer devolver un objeto como respuesta, podemos emplear el método `json` de la respuesta (más adelante veremos que todos los objetos emitidos directamente al cliente se envían en formato JSON), y así podremos adjuntar un código de estado diferente de 200:

```
return response()->json(['datos' => datos], 201)
    ->header('Cabecera1', 'Valor1')
    ...;
```

2.2.1. Usar la respuesta para hacer redirecciones

Existe también un método `redirect` que podemos emplear para redireccionar a una ruta desde otra, bien especificando la ruta como parámetro...

```
redirect('/');
```

... o bien indicando una ruta con nombre:

```
redirect()->route('inicio');
```

Podemos pasar valores a la siguiente redirección, almacenándolos en sesión con el método `with`, aunque estos valores se perderán en la siguiente petición (no se quedan almacenados en sesión):

```
redirect()->route('inicio')
    ->with('mensaje', 'Mensaje enviado correctamente');
```

Para acceder a este mensaje desde la vista afectada, debemos utilizar la función `session`:

```
@if(session()->has('mensaje'))
    {{ session('mensaje') }}
@endif
```

Por último, notar que si hacemos la redirección desde dentro de un método de un controlador (por ejemplo, para redigir a una ruta desde otra), deberemos *devolver* (`return`) el resultado de esa redirección para que surta efecto:

```
class LibroController extends Controller
{
    public function index()
    {
        ...
    }

    public function store(...)
    {
        ...
        return redirect()->route('libros.index');
    }
```

2.3. Ejemplo: helpers

Para terminar esta introducción a lo que supone la inyección de dependencias en frameworks de desarrollo, vamos a hacer uso de una herramienta que nos puede ser útil en algunas situaciones: los *helpers*.

Un **helper** es básicamente una función de utilidad que podemos querer utilizar en diversos puntos de nuestra web, y que necesitamos tener localizada y compartida. Por ejemplo, imaginemos que queremos resaltar en nuestro menú de navegación la opción que tenemos actualmente visible.

Para ello, podemos definir una clase CSS con el estilo que queramos para resaltar (esto lo haremos aparte, en los archivos CSS del proyecto), y después utilizar esa clase CSS en una condición para cada menú de navegación.

Por ejemplo, supongamos que la clase CSS para identificar el menú activo se llama `activo`. En este caso, para un menú de varias opciones como éste, basta con utilizar el método `routeIs` de la petición (`request`) para comprobar si la ruta coincide con cada menú, y mostrarlo como activo o no, usando un operador ternario de comparación:

```
<nav>
    <ul>
        <li class="{{ request()->routeIs('inicio') ? 'activo' : '' }}">
            <a href="/">Inicio</a>
        </li>
        <li class="{{ request()->routeIs('contacto') ? 'activo' : '' }}">
            <a href="/contacto">Contacto</a>
        </li>
        ...
    </ul>
</nav>
```

Esta característica también funciona si las rutas tienen parámetros.

Podemos, en cambio, sacar fuera de la vista la lógica de establecer un campo como activo o no. Para ello, creamos un archivo de utilidad o *helper*. Lo podemos llamar `helpers.php`, y ubicarlo en la misma carpeta `app`. Dentro, definimos la función que nos va a devolver si una ruta está activa o no, a partir de su nombre:

```
function setActivo($nombreRuta)
{
    return request()->routeIs($nombreRuta) ? 'activo' : '';
}
```

Y de este modo, nuestra vista simplemente se dedica a llamar a esta función para cada elemento del menú:

```
<nav>
    <ul>
        <li class="{{ setActivo('inicio') }}">
            <a href="/">Inicio</a>
        </li>
        <li class="{{ setActivo('contacto') }}">
            <a href="/contacto">Contacto</a>
        </li>
        ...
    </ul>
</nav>
```

En el caso de querer mantener el enlace activo para cualquier subruta a partir de la original (por ejemplo, cuando estamos viendo la ficha de un registro a partir del listado general, podemos utilizar el *wildcard* de asterisco `*`):

```
<li class="{{ setActivo('peliculas.*') }}">
    <a href="{{ route('peliculas') }}">Peliculas</a>
</li>
```

Sin embargo, para que Laravel cargue el archivo `helpers.php` que acabamos de crear, como no es una clase, debemos indicarlo explícitamente (Laravel carga automáticamente todas las clases de la carpeta `app`, pero no archivos sueltos que no sean clases). Para ello, debemos ir al archivo `composer.json` de la raíz de nuestro proyecto, a la sección `autoload` y añadir una sección `files` con un array con los archivos que queramos que se carguen también:

```
"autoload": {
    "classmap": [ ... ],
    "psr-4": { ... },
    "files": ["app/helpers.php"]
},
```

Tras efectuar el cambio, debemos decirle a *composer* que vuelva a compilar el auto cargador. Desde la carpeta del proyecto, ejecutamos este comando:

```
composer dump-autoload
```

En este punto, puedes realizar el Ejercicio 2 de los propuestos al final de la sesión.

3. Ejercicios propuestos

Ejercicio 1

Sobre el proyecto **blog** de la sesión anterior, vamos a añadir estos cambios:

- Crea un controlador de recursos (opción `-r`) llamado `PostController`, que nos servirá para gestionar toda la lógica de los posts del blog.
- Asigna automáticamente con el método `resource` cada ruta a su función correspondiente del controlador, en el archivo `routes/web.php`. Limita con `only` las acciones sólo a las funciones de listado (`index`), ficha (`show`), creación (`create`) y edición (`edit`).
- Utiliza el proveedor de servicios `AppServiceProvider` para "castellanizar" las rutas de creación y edición, como en el ejemplo que hemos visto de libros.
- Renombra las vistas de listado y ficha de un post a `index.blade.php` y `show.blade.php`, dentro de su carpeta `posts`, y haz que los métodos correspondientes del controlador de posts rendericen estas vistas. Para los métodos `create` y `edit`, simplemente devuelve un texto plano indicando "Nuevo post" y "Edición de post", por ejemplo.

- Haz los cambios adicionales que sean convenientes (por ejemplo, en el menú de navegación) para que los enlaces sigan funcionando, y prueba que las cuatro rutas (listado, ficha, creación y edición) funcionan adecuadamente.

Ejercicio 2

Sobre el proyecto **blog** anterior, vamos a añadir estos cambios:

- Haz que las funciones de `create` y `edit` del controlador de posts, en lugar de mostrar un mensaje de texto plano indicando que ahí va un formulario, redirijan a la página de inicio, usando la instrucción `redirect`.
- Añade un *helper* al proyecto que defina una función llamada `fechaActual`. Recibirá como parámetro un formato de fecha (por ejemplo, "d/m/y") y sacará la fecha actual en dicho formato. Utilízalo para mostrar la fecha actual en formato "d/m/Y" en la plantilla base, bajo la barra de navegación, alineada a la derecha.

¿Qué entregar?

Como entrega de esta sesión deberás comprimir el proyecto **blog** con todos los cambios incorporados, y eliminando las carpetas `vendor` y `node_modules` como se explicó en las sesiones anteriores. Renombra el archivo comprimido a `blog_03.zip`.