Sesión 4. El modelo y los datos (I): migraciones y modelos simples

En esta sesión comenzaremos a ver algunas cuestiones importantes sobre cómo gestiona Laravel el acceso a bases de datos, y qué mecanismos ofrece para sincronizar los datos de nuestra aplicación con los documentos o registros de una base de datos, así como para generar automáticamente la estructura de tablas y campos de la base de datos a partir del modelo de la aplicación.

1. Parámetros de conexión a la base de datos

Una de las primeras cosas que debemos hacer para configurar el acceso a la base de datos en nuestro proyecto es establecer los parámetros con los que conectar a dicha base de datos: nombre del servidor, usuario, contraseña, etc. Estos parámetros se definen en el archivo `.env` para cada entorno despliegue de la aplicación (recuerda que este archivo no se sube a Git, por lo que cada entorno tendrá el suyo). Dentro de este archivo, debemos modificar las siguientes variables de entorno:

- `DB_CONNECTION` : tipo de SGBD a usar
- `DB_HOST` : dirección o IP del SGBD (*127.0.0.1* para conexión local)
- `DB_PORT` : puerto por el que el SGBD estará escuchando. Por ejemplo, el puerto por defecto para MySQL es 3306
- `DB_DATABASE` : nombre de la base de datos a la que conectar
- `DB_USERNAME` : login del usuario para conectar
- `DB_PASSWORD` : password del usuario para conectar

En cuanto al primer parámetro (`DB_CONNECTION`), aquí tenemos un listado de los sistemas más habituales, junto con sus puertos por defecto que podemos utilizar en `DB_PORT` :

Id SGBD	Nombre SGBD	Puerto por defecto
mysql	MySQL/MariaDB	3306
oracle	Oracle	1521
pgsql	PostgreSQL	5432
sqlsrv	SQL Server	1433
sqlite	SQLite	-

Por ejemplo, para nuestro ejemplo de la biblioteca, el archivo `.env` del proyecto podría quedar así, suponiendo el usuario y contraseña por defecto que se instala con XAMPP (usuario *root* y password vacío).

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=biblioteca
DB_USERNAME=root
DB_PASSWORD=
```

En el archivo `config/database.php` existen unos valores por defecto asociados a cada parámetro de configuración del archivo `.env`, de modo que si no se encuentra el parámetro, se toma el valor por defecto. Por ejemplo, el SGBD seleccionado si no se especifica ninguno es *mysql*, a juzgar por esta línea del archivo `database.php`:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

1.1. Creación de la base de datos

El único paso necesario desde fuera de Laravel para acceder a la base de datos será crearla. El resto de operaciones (creación de tablas, campos, claves, relaciones, datos, etc) se podrán hacer desde el propio Laravel, como iremos viendo a continuación.

La base de datos podemos crearla a través de algún administrador que tengamos disponible (por ejemplo, *phpMyAdmin* para bases de datos MySQL), o bien por línea de comandos, conectando con el SGBD en cuestión y creando la base de datos.

En nuestro caso, tendremos que crear una base de datos llamada "biblioteca", tal y como hemos especificado en la propiedad `DB_DATABASE` del archivo `.env`. Vamos a la opción *Nueva* del panel izquierdo y escribimos el nombre de la nueva base de datos en el formulario que aparecerá. Pulsando el botón de *Crear* ya aparecerá la nueva base de datos en el listado izquierdo.

2. Las migraciones

Las migraciones suponen una especie de control de versiones para una base de datos, y permiten crear y modificar el esquema de dicha base de datos fácilmente.

2.1. Estructura de las migraciones

Por defecto, Laravel trae unas migraciones predefinidas, que se hallan en la carpeta `database/migrations`. Cada una tiene un nombre de archivo que comienza por la fecha en que se hizo, seguida de una breve descripción de lo que contiene (creación de la tabla de usuarios, reseteo de contraseñas...). Puede que algunas de estas migraciones no nos vayan a ser necesarias, con lo que podemos borrarlas directamente, y puede que otras (en especial la creación de la tabla de usuarios) sí nos sirva, pero con otros campos, con lo que deberemos editarla, como veremos a continuación.

Si examinamos el contenido de una migración, todas deben tener dos métodos:

- `up` : permite agregar tablas, columnas o índices a la base de datos
- `down` : revierte lo hecho por el método anterior

Si observamos el contenido de un método `up` de los que vienen predefinidos para crear una tabla, vemos que se utilizan distintos métodos para definir los tipos de datos de cada campo de la tabla, como por ejemplo `id()` para campos que puedan contener enteros autoincrementales, o `string()` para campos de tipo texto. Además, existen otros métodos modificadores para agregar propiedades adicionales, como por ejemplo `unique()` para indicar valores únicos (claves alternativas), o `nullable()` para indicar que un campo admite nulos. Aquí tenemos un ejemplo de método `up` :

```
public function up()
{
    Schema::create('usuarios', function(BluePrint $tabla) {
        $tabla->id();
        $tabla->string('nombre');
        $tabla->string('email')->unique();
        ...
        $tabla->timestamps();
    });
}
```

Por defecto, como vemos en los ejemplos que se proporcionan, los esquemas se crean con un *id* autonumérico, y unos *timestamps* para indicar la fecha de creación y de modificación de cada registro, y que Laravel gestiona de forma automática cuando insertamos o actualizamos contenidos, lo cual resulta muy útil.

Sobre esta base, podemos añadir o quitar los campos que queramos. Para ver los tipos disponibles para las columnas de la tabla, podemos visitar la documentación de Laravel sobre migraciones, en concreto buscaremos el subapartado *Available Column Types*. Conviene tener presente, por ejemplo, que el tipo

`string` que hemos utilizado en el ejemplo anterior tiene una limitación de 255 caracteres. Para textos más grandes, se puede emplear el tipo `text` (20.000 caracteres aproximadamente) o `longText`.

Podemos especificar una clave primaria con el método `primary`, al que le podemos pasar o bien el nombre del campo clave, o un array de campos clave, en el caso de que ésta sea compuesta. Por defecto, los campos de tipo `id` se auto-establecen como claves primarias.

```
$table->primary(['campo1', 'campo2']);
```

2.2. Creación de migraciones

Creamos migraciones con el comando:

```
php artisan make:migration nombre_migracion
```

Por ejemplo:

```
php artisan make:migration crear_tabla_prueba
```

Notar que Laravel ya asigna automáticamente la fecha de la migración, sólo debemos especificar el nombre descriptivo de la misma. Además, si Laravel detecta la palabra *create* en el nombre de la migración, finalizada en *table*, intuye que es para crear una tabla nueva. En cambio, si detecta la palabra *to* (entre otras), y al final la palabra *table*, intuye que se va a alterar o modificar una tabla existente. Esto es gracias a la clase `TableGuesser` incorporada en Laravel, que detecta ciertos patrones en los nombres de migraciones. La diferencia entre la creación y la modificación es que en el método `up` de la migración se utilizará `Schema::create` o `Schema::table` sobre la tabla en cuestión, respectivamente.

En cualquier caso, también podemos especificar un parámetro adicional en el comando de migración para indicar si queremos crear o modificar una tabla, y de este modo podemos definir el nombre de la migración en el idioma que queramos, y sin restricciones de patrones. Estas dos migraciones crean una tabla (*pedidos*) y modifican otra (*usuarios*), respectivamente:

```
php artisan make:migration crear_tabla_pedidos --create=pedidos
php artisan make:migration nuevo_campo_usuario --table=usuarios
```

En el caso de la segunda migración, si, por ejemplo, queremos añadir una columna con el número de teléfono de los usuarios, puede quedar así (tanto el método `up` como el `down`):

```
public function up()
{
    Schema::table('usuarios', function(Blueprint $tabla) {
        $tabla->string('telefono')->nullable();
    });
}

public function down()
{
    Schema::table('usuarios', function(Blueprint $tabla) {
        $tabla->dropColumn('telefono');
    });
}
```

Si queremos que el campo en cuestión esté en un orden concreto, podemos usar el método `after` para indicar detrás de qué campo queremos ponerlo (en el método `up`):

```
$tabla->string('telefono')->after('email')->nullable();
```

2.3. Ejecución y borrado de migraciones

Para ejecutar las migraciones (el método `up` de cada una), lanzamos el siguiente comando desde la carpeta de nuestro proyecto (habiendo creado la base de datos ya previamente, y modificado las credenciales de acceso en el archivo `.env`):

```
php artisan migrate
```

Adicionalmente a las tablas afectadas, se tendrá otra tabla `migrations` en la base de datos con un histórico de las migraciones realizadas. Para cada una, se almacena su *id* (autonumérico), el nombre de la migración, y el número de proceso por lotes en que se hizo (aquellas que compartan el mismo número se hicieron a la vez en el mismo lote). De este modo, aquellas que ya se hayan hecho no se volverán a realizar.

Para deshacer las migraciones realizadas (ejecutar el método `down` de las mismas), ejecutamos:

```
php artisan migrate:rollback
```

Esto eliminará TODAS las migraciones del último lote existente en la tabla `migrations`. Si no queremos deshacerlo todo, sino retroceder un número determinado de migraciones dentro de ese lote, ejecutamos el comando anterior con un parámetro `--step`, indicando el número de pasos o migraciones a deshacer (en orden cronológico de más reciente a más antigua):

```
php artisan migrate:rollback --step=2
```

Si volvemos a hacer la migración, se restablecerán las migraciones deshechas de ese lote.

Otro comando también muy utilizado es `migrate:fresh`. Lo que hace es eliminar todas las migraciones realizadas y volverlas a lanzar. Es útil cuando, estando en desarrollo, añadimos campos nuevos a alguna tabla y queremos rehacer las tablas completamente.

```
php artisan migrate:fresh
```

> **NOTA**: el comando `migrate:fresh` es DESTRUCTIVO, elimina los contenidos de las tablas, y sólo debe utilizarse en entornos de desarrollo, no de producción.

2.4. Aplicando las migraciones a nuestro ejemplo

Vamos a poner en práctica todo lo visto en este apartado sobre nuestro proyecto `biblioteca`.

1. En el punto 1 de esta sesión ya hemos comentado cómo modificar el archivo `.env` del proyecto para darle los parámetros de conexión correctos a la base de datos, y cómo crear la base de datos "biblioteca" desde *phpMyAdmin*. Revisa ese apartado para hacer estos pasos, si no los has hecho ya.
2. A continuación, vamos a eliminar las migraciones que no nos van a ser necesarias de la carpeta `database/migrations`. En concreto, borramos las que hacen referencia a `create_password_resets_table` y a `create_failed_jobs_table`.
3. Después, editamos la migración para la tabla de usuarios (`create_users_table`), ya que la utilizaremos en sesiones posteriores. Podemos renombrar el archivo a `crear_tabla_usuarios`. La clase interna también la renombramos a `CrearTablaUsuarios`, y la tabla a la que se alude en los métodos `up` y `down` también la renombramos a `usuarios`, para dejarlo en nuestro idioma (respetando la fecha de creación en el nombre del archivo), y después editamos el método `up` para dejarlo así:

```
public function up()
{
    Schema::create('usuarios', function(Blueprint $table) {
        $table->id();
        $table->string('login')->unique();
        $table->string('password');
        $table->timestamps();
    });
}
```

4. Ahora vamos a crear una nueva migración para definir la estructura de los libros:

```
php artisan make:migration crear_tabla_libros --create=libros
```

5. Editamos después el contenido de esta migración, en concreto el método `up` para definir estos campos en los libros:

```
public function up()
{
    Schema::create('libros', function(Blueprint $table) {
        $table->id();
        $table->string('titulo');
        $table->string('editorial')->nullable();
        $table->float('precio');
        $table->timestamps();
    });
}
```

6. Cargamos las migraciones con el comando:

```
php artisan migrate
```

Tras esto, ya deberíamos ver en nuestra base de datos "*biblioteca*" las dos tablas creadas (*usuarios* y *libros*), junto con la tabla *migrations* que crea Laravel para gestionar las migraciones realizadas.

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

3. Gestión del modelo de datos

Ahora que ya tenemos la estructura de tablas creada en la base de datos, vamos a ver qué mecanismos ofrece Laravel para acceder a estos datos de forma sencilla desde la aplicación. Veremos cómo definir el modelo de datos asociado a cada tabla, y cómo manipular estos datos empleando el ORM Eloquent, incorporado con Laravel.

3.1. Crear el modelo

La idea es crear una clase por cada tabla que tengamos en nuestra base de datos, para así interactuar con la tabla a través de dicha clase asociada. Para crear esta clase modelo, utilizamos la opción `make:model` del comando `php artisan`. Le pasaremos como parámetro adicional el nombre de la clase a crear. Por ejemplo, para el caso de nuestra biblioteca, podemos crear así el modelo `Libro`:

```
php artisan make:model Libro
```

Por convención, los modelos se crean con un nombre en singular, empezando por mayúscula, y se ubican en la carpeta `app\Models`. La estructura básica del modelo es algo así:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Libro extends Model
{

}

?>
```

En nuestro caso, vamos también a utilizar el modelo de usuario que ya existe en la carpeta `app\Models`, aunque lo renombraremos de `User` a `Usuario`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Usuario extends Model
{
    ...
```

> **NOTA**: hasta Laravel 7, los modelos se generaban automáticamente en la carpeta `app`, y era necesario moverlos manualmente a una subcarpeta si queríamos estructurar mejor nuestro código, actualizando también el `namespace` correspondiente. En Laravel 8 la ubicación en la carpeta `app\Models` se realiza por defecto.

Automáticamente, se asocia este modelo a una tabla con el mismo nombre, pero en plural y en minúscula, por lo que los modelos anteriores estarían asociados a unas tablas *libros* y *usuarios* en la base de datos, respectivamente. En caso de que no queramos que sea así, definimos una propiedad `$table` en la clase con el nombre que queramos que tenga la tabla asociada. Por ejemplo:

```
class Libro extends Model
{
    protected $table = 'mislibros';
}
```

3.1.1. Otras opciones de crear modelos

El comando anterior `make:model` admite unos parámetros adicionales, de forma que se puede crear a la vez el modelo y la migración, y más aún, el modelo, la migración y el controlador asociado. Veamos algunos ejemplos:

```
php artisan make:model Pelicula -m
```

El comando anterior crea un modelo `Pelicula` en la carpeta `app\Models` y, además, crea una migración llamada `create_peliculas_table` en la carpeta `database/migrations`, lista para que editemos el método `up` y especifiquemos los campos necesarios.

Notar que el nombre de la migración añade una "s" al nombre de la tabla automáticamente, a partir del modelo en singular.

```
php artisan make:model Pelicula -mc
```

Este otro comando crea lo mismo que el anterior, y además, un controlador llamado `PeliculaController` en la carpeta `app\Http\Controllers`. Dicho controlador está vacío, para que añadamos los métodos que consideremos.

```
php artisan make:model Pelicula -mcr
```

Esta otra opción crea lo mismo que la anterior, pero el controlador `PeliculaController` es en este caso un controlador de recursos, por lo que tiene ya incorporados el conjunto de métodos propios de este tipo de controladores: `index`, `show`, etc.

Podemos también usar la versión extendida de estos parámetros. Por ejemplo:

```
php artisan make:model Pelicula --migration --controller --resource
```

En nuestro caso, como hemos ido creando los controladores y migraciones antes que los modelos, no sería necesario dar este paso, pero ahora que ya empezamos a ver cómo funciona y se interrelaciona todo, puede

resultar útil emplear este comando para crear de golpe todas las partes implicadas (modelo, migración y controlador)

3.1.2. Seguir una nomenclatura uniforme

Recuerda que, de sesiones anteriores, hemos comentado la recomendación/necesidad de seguir una nomenclatura uniforme en los modelos, controladores y vistas. Así, para el modelo `Libro` ya tendríamos su controlador asociado `LibroController`, y las vistas se definirían en la subcarpeta `resources/views/libros`, con los nombres correspondientes a cada método del controlador (por ejemplo, `index.blade.php`, o `show.blade.php`).

3.2. Operaciones sobre el modelo. Primeros pasos con Eloquent

Eloquent es el ORM incorporado por defecto en Laravel. Un ORM (*Object Relational Mapping*) es una herramienta que permite establecer una relación entre los registros de una tabla de la base de datos y los objetos de una clase (de PHP en nuestro caso), de forma que los datos de la base de datos se convierten a objetos PHP y viceversa. Además, Eloquent implementa el patrón *Active Record*, que añade a las clases métodos como `save`, `update`, `delete`... que permiten interactuar con la base de datos para insertar, modificar o borrar registros asociados a objetos, respectivamente.

3.2.1 Realizar búsquedas

Una vez creado el modelo, y aunque esté vacío, ya podemos utilizarlo en los controladores para acceder a los datos. Basta con importar la clase correspondiente (con `use`), y utilizar los métodos que se heredan de `Model`. Por ejemplo, el método `get` permite obtener los registros de la tabla, convertidos a objetos. Así es como obtendríamos todos los libros de la tabla desde un controlador:

```
...
use App\Models\Libro;
...

class LibroController extends Controller
{
    public function index()
    {
        $libros = Libro::get();
        return view('libros.index', compact('libros'));
    }
}
```

Lo que obtenemos es un array de objetos, por lo que deberemos acceder a sus propiedades como tales. Por ejemplo, si queremos mostrar los títulos de los libros en una vista Blade, haríamos algo como esto:

```
@forelse($libros as $libro)
    {{ $libro->titulo }}
@endforelse
```

Alternativamente, también podemos obtener una **consulta filtrada**, especificando con el método `where` la condición que deben cumplir los registros a obtener. Por ejemplo, así obtendríamos los libros cuyo precio sea inferior a 10 euros:

```
$libros = Libro::where('precio', '<', 10)->get();
```

De este otro modo obtendríamos libros con precio inferior a 10 euros y superior a 5 euros, de modo que podemos combinar condiciones:

```
$libros = Libro::where('precio', '<', 10)
    ->where('precio', '>', 5)->get();
```

Sobre estas consultas base podemos aplicar una serie de añadidos. Por ejemplo, podemos querer ordenar los libros por título, para lo que haríamos esto en el controlador:

```
$libros = Libro::orderBy('titulo')->get();
```

El método `orderBy` admite un segundo parámetro que indica el sentido de la ordenación. Por defecto es `ASC` (ascendente), pero también puede ser `DESC`:

```
$libros = Libro::orderBy('titulo', 'DESC')->get();
```

Paginaciones de resultados

Si queremos paginar los resultados obtenidos, debemos, por un lado, cuando obtengamos el listado desde el controlador, indicar con `paginate` cuántos registros queremos por página:

```
public function index()
{
    $libros = Libro::paginate(5);
    return view('libros.index', compact('libros'));
}
```

Después, en la vista asociada (`libros.index` en el ejemplo anterior), podemos emplear el método `links` para que muestre los botones de paginación en el lugar deseado:

```
@forelse($libros as $libro)
    {{ $libro->titulo }}
@endforelse

{{ $libros->links() }}
```

Si queremos ordenar el listado, podemos emplear "orderBy" u "orderByDesc", pasándole como parámetro el nombre del campo por el que ordenar, antes de la paginación. Podemos, incluso, ordenar por múltiples criterios concatenados:

```
public function index()
{
    $libros = Libro::orderByAsc('titulo')
        ->orderByAsc('editorial')
        ->paginate(5);
    return view('libros.index', compact('libros'));
}
```

Paginaciones desde Laravel 8

En la versión 8 de Laravel se ha cambiado el estilo de los botones de paginación, empleando el del framework Tailwind CSS. Si queremos seguir utilizando los de Bootstrap, debemos añadir esta línea en el método `boot` del *provider* `App\Providers\AppServiceProvider`:

```
Paginator::useBootstrap();
```

Además, debemos incorporar la cláusula `use` para localizar el elemento `Paginator`:

```
use Illuminate\Pagination\Paginator;
```

3.2.2. Fichas de objetos individuales

Una operación bastante habitual es mostrar una ficha de un objeto a partir de un listado, haciendo clic en el título o alguna parte visible de ese objeto. Por ejemplo, si queremos ver los datos de un libro a partir de un listado con sus títulos, podemos hacer algo como esto en la plantilla Blade:

```
@forelse($libros as $libro)
    <li><a href="{{ route('libros.show', $libro) }}">
            {{ $libro->titulo }}
    </a></li>
@endforelse
```

Vemos que hemos utilizado el método `route` para indicar la ruta a seguir, con un segundo parámetro, que en este caso es el objeto concreto de esa fila. Laravel automáticamente lo reemplazará en el enlace por el identificador de dicho objeto.

Por su parte, la ruta asociada a este enlace podría ser algo así (en el archivo de rutas):

```
Route::get('/libros/{id}', [LibroController::class, 'show'])
->name('libros.show');
```

Aunque también podemos haber definido las rutas como un paquete de recursos, y cada una tendrá su método asociado:

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

Finalmente, el método `show` del controlador asociado se encargará de obtener los datos del libro a partir de su *id*, y generar la vista correspondiente. Para obtener los datos de un objeto a partir de su identificador, podemos emplear el método `find` del modelo, pasándole como parámetro el identificador. Así, podríamos generar una vista con los datos como ésta:

```
...
class LibroController extends Controller
{
    ...

    public function show($id)
    {
        $libro = Libro::find($id);
        return view('libros.show', compact('libro'));
    }
}
```

> **NOTA**: si devolvemos (`return`) directamente lo que obtiene el método `find`, nos llegará al navegador en formato JSON. De hecho, si devolvemos un array, Laravel lo envía directamente en formato JSON. Esta característica la utilizaremos más adelante para definir servicios REST.

En el caso de que el objeto no se encuentre (porque, por ejemplo, utilicemos un *id* equivocado), la vista generada fallará. Para evitarlo, en lugar del método `find` podemos emplear `findOrFail`, que, en caso de que no se encuentre el objeto, generará una vista con un error 404, más apropiada. Además, recuerda que puedes personalizar estas páginas de error definiendo las vistas correspondientes.

```
$libro = Libro::findOrFail($id);
```

En este punto, y a falta de que podamos hacer inserciones más adelante, puedes probar a insertar unos pocos libros de prueba en la base de datos *biblioteca* desde phpMyAdmin, y probar estas dos rutas que hemos hecho (listado y ficha de libro).

3.2.3. Inserciones

Las inserciones a través de Eloquent se pueden realizar creando una instancia del objeto, rellenando sus atributos y llamando al método `save`, heredado de la superclase `Model`.

```
$libro = new Libro();
$libro->titulo = "El juego de Ender";
$libro->editorial = "Ediciones B";
$libro->precio = 8.95;
$libro->save();
```

Como alternativa, también se puede utilizar el método `create` del modelo, y pasarle todos los datos de la petición, que llegarían desde un formulario, como veremos más adelante:

```
Libro::create($request->all());
```

Para que esto último funcione, deben cumplirse dos premisas:

- Cada campo de la petición debe tener asociado un campo del mismo nombre en el modelo.
- Debemos definir en el modelo una propiedad llamada `$fillable` con los nombres de los campos de la petición que nos interesa procesar (el resto se descartan). Esto es obligatorio especificarlo, aunque nos interesen todos los campos, para evitar inserciones masivas malintencionadas (por ejemplo, editando el código fuente para añadir otros campos y modificar datos inesperados).

```
class Libro extends Model
{
    protected $fillable = ['titulo', 'editorial', 'precio'];
}
```

Este código de inserción (o bien campo a campo, o usando el método `all`) se suele poner en el método `store` del controlador, para que reciba los datos del formulario de inserción y la haga en la base de datos. Lo terminaremos de ver cuando abordemos el tema de los formularios en Laravel.

3.2.4. Modificaciones

La modificación consiste en dos pasos:

- Encontrar el objeto a modificar (buscándolo por el *id* con `findOrFail`, por ejemplo, como se ha explicado antes)
- Modificar las propiedades que se necesiten, y llamar al método `save` del objeto para guardar los cambios.

Por ejemplo:

```
$libroAModificar = Libro::findOrFail($id);
$libroAModificar->titulo="Otro título";
$libroAModificar->save();
```

También podemos utilizar el método `update` enlazado con `findOrFail`, y pasarle como parámetro todos los datos de la petición, igual que se ha explicado para la inserción, y siempre y cuando hayamos declarado el atributo `$fillable` en el modelo para indicar qué campos se aceptan:

```
Libro::findOrFail($id)->update($request->all());
```

Este código de modificación se suele poner en el método `update` del controlador, para que reciba los datos del formulario de edición y haga la modificación correspondiente. Lo terminaremos de ver cuando abordemos el tema de los formularios en Laravel.

3.2.5. Borrados

Para hacer el borrado, también buscamos el objeto a borrar con `findOrFail`, y luego llamamos a su método `delete`:

```
Libro::findOrFail($id)->delete();
```

Esto lo haremos normalmente en el método `destroy` del controlador en cuestión. Después, podemos redirigir o renderizar alguna vista resultado, como el listado de libros general para comprobar que se ha borrado.

```
public function destroy($id)
{
    Libro::findOrFail($id)->delete();
    $libros = Libro::get();
    return view('libros.index', compact('libros'));
}
```

Sobre el borrado desde las vistas

Lo normal es que el borrado se active haciendo clic en algún elemento de una vista. Por ejemplo, haciendo clic en un botón o enlace que ponga "Borrar". Sin embargo, si implementamos esto así:

```
<a href="{{ route('libros.destroy', $libro }}">
Borrar
</a>
```

Si queremos borrar el libro con *id* 3, se generará una ruta *http://biblioteca/libros/3*. Lo podemos comprobar pasando el ratón por el enlace y viendo la barra inferior de estado del navegador. Esta ruta, sin embargo, nos va a enviar a la ficha del libro 3, no al borrado, ya que estamos enviando una petición GET, y no una de borrado (DELETE). Para evitar esto, la opción de borrado debe hacerse siempre desde un formulario, donde a través del helper `@method` indicamos que es una petición de borrado (DELETE). Con lo que el "enlace" para borrar un libro quedaría así:

```
<form action="{{ route('libros.destroy', $libro) }}" method="POST">
    @method('DELETE')
    @csrf
    <button>Borrar</button>
</form>
```

> **NOTA** el helper `@csrf` lo veremos con más detalle al hablar de formularios, pero se añade a los formularios Laravel para evitar ataques de tipo *cross-site*, es decir, accesos a una URL de nuestra web desde otras webs.

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

4. Ejercicios propuestos

Ejercicio 1

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

- Crea una base de datos llamada `blog` en tu servidor de bases de datos a través de *phpMyAdmin*. Modifica también el archivo `.env` del proyecto para acceder a dicha base de datos con las credenciales adecuadas, similares a las del ejemplo de la biblioteca (cambiando el nombre de la base de datos).
- Elimina las migraciones relativas a *password_resets* y *failed_jobs*, y edita la migración de la tabla usuarios para dejarla igual que el ejemplo de la biblioteca (únicamente con los campos *login* y *password*, además del *id* y los *timestamps*).
- Crea una nueva migración llamada `crear_tabla_posts`, que creará una tabla llamada `posts` con estos campos:
    - Id autonumérico
    - Título del post (`string`)
    - Contenido del post (`text`)
    - *Timestamps* para gestionar automáticamente la fecha de creación o modificación del post
- Lanza las migraciones y comprueba que se crean las tablas correspondientes con los campos asociados en la base de datos.

Ejercicio 2

Continuamos con el proyecto **blog** anterior. Crea un nuevo modelo llamado `Post` para los posts de nuestro blog. Muévelo junto con el modelo de `Usuario` a la subcarpeta `App\Models` del proyecto.

Después, modifica los métodos del controlador `PostController` creado en sesiones anteriores, de este modo:

- El método `index` debe obtener todos los posts de la tabla, y mostrar la vista `posts.index` con ese listado de posts.
    - La vista `posts.index`, por su parte, recibirá el listado de posts y mostrará los títulos de cada uno, y un botón `Ver` para mostrar su ficha (`posts.show`).
    - Debes mostrar el listado de posts ordenado por *título* en orden ascendente, y paginado de 5 en 5.
- El método `show` debe obtener el post cuyo *id* se pasará como parámetro, y mostrarlo en la vista `posts.show`.
    - La vista `posts.show` recibirá el objeto con el post a mostrar, y mostraremos el título, contenido y fecha de creación del post, con el formato que quieras.
- El método `destroy` eliminará el post cuyo *id* recibirá como parámetro, y devolverá la vista `posts.index` con el listado actualizado. Para probar este método, recuerda que debes definir un formulario en una vista (lo puedes hacer para cada post mostrado en la vista `posts.index`) que envíe a la ruta `posts.destroy` usando un método *DELETE*, como hemos explicado en un ejemplo anterior.
- Los métodos `create`, `edit`, `store` y `update` de momento los vamos a dejar sin hacer, hasta que veamos cómo gestionar formularios.

- Para simular la inserción y la modificación, vamos a crear dos métodos adicionales en el controlador, que usaremos de forma temporal:
  - Un método llamado `nuevoPrueba`, que cada vez que lo llamemos creará un post con un título al azar (por ejemplo, "Título X", siendo X un entero aleatorio), y un contenido al azar ("Contenido X"). Puedes emplear la función rand de PHP para generar estos números aleatorios para título y contenido.
  - Un método llamado `editarPrueba`, que recibirá como parámetro un *id* y modificará el título y contenido del post otros generados aleatoriamente, como en el punto anterior.
  - Estos dos métodos (especialmente el primero) nos servirán para crear una serie de posts de prueba que luego nos servirán para probar el listado y la ficha de los posts.

- En el archivo `routes/web.php`, recuerda añadir dos nuevas rutas temporales de tipo `get` para probar estas inserciones y modificaciones. La primera puede apuntar a `/libros/nuevoPrueba`, por ejemplo, y la segunda a `/libros/editarPrueba/{id}`. Recuerda también eliminar o editar la restricción `only` de las rutas del controlador que estableciste la sesión anterior, para que no sólo permita las rutas *index*, *show*, *create* y *edit*, y además permita la de *destroy* (o todas las posibles, si quieres, ya que tarde o temprano las utilizaremos).

¿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_04.zip`.