El 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.

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
<?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
<?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. Desde 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:

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

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)

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`).

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.

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;
```

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).

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.

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.

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.