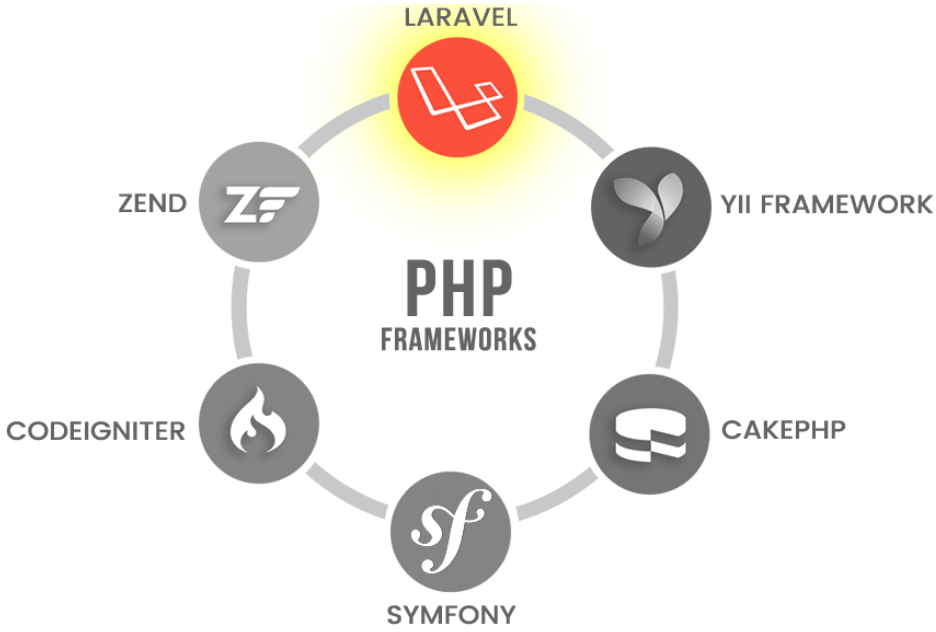
unidad didáctica 7

Laravel - modelo de datos



1. acceso a la base de datos

Una vez vistas dos de las tres patas en que se sustenta el patrón MVC (las vistas y los controladores), en esta sección abordaremos la tercera de ellas: el modelo de datos. Con esto, trataremos 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.

1.1. 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 de 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'),
```

1.2. 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 más adelante.

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. Para la máquina virtual que estamos utilizando, podemos acceder a *phpMyAdmin* teniendo XAMPP en marcha (tanto el servidor Apache como el de MySQL) y accediendo a la URL *http://localhost/phpmyadmin*, normalmente.

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. migraciones

Las migraciones son un mecanismo de definición de datos ofrecido por Laravel para, a través de ciertas clases y opciones de configuración, generar la estructura completa de una base de datos. A su vez, suponen una especie de control de versiones para una base de datos, y permiten crear y modificar el esquema de la misma 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`:

```php
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`):

```php
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`):

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

2.2.1. 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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.

2.2.2. Aplicando las migraciones a nuestro ejemplo

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

A continuación, vamos a eliminar las migraciones que no nos van a ser necesarias de la carpeta `database/migrations`. En concreto, borramos todas salvo la de creación de la tabla de usuarios `create_users_table`.

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 en versiones recientes de Laravel no tiene nombre, se crea simplemente un subtipo de *Migration*:

```
// ...

return new class extends Migration
{
  // ...
}
```

NOTA: en versiones anteriores donde sí tenga nombre, podemos reemplazar el nombre viejo (*CreateUsersTable*) por *CrearTablaUsuarios*, por ejemplo.

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();
  });
}
```

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

```
php artisan make:migration crear_tabla_libros --create=libros
// ó, si no funciona:
// sudo docker-compose exec myapp php artisan make:migration
crear_tabla_libros --create=libros
```

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();
  });
}
```

Cargamos las migraciones con el comando:

```
php artisan migrate
// ó, si no funciona:
// sudo docker-compose exec myapp 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.

NOTA: si, además de las tablas anteriores, aparece una tabla de *personal_access_tokens* (ésta la crea automáticamente el paquete *Sanctum*) incorporada por defecto en Laravel. No nos molesta para lo que vamos a hacer durante el curso, pero si queréis quitarla, hay que editar el archivo `app\Providers\AppServiceProvider.php` y añadir esta línea en el método `register` (junto con el correspondiente `use` para la clase `Sanctum`):

```
// ...
use Laravel\Sanctum\Sanctum;

class AppServiceProvider ...
{
  // ...
  public function register()
  {
      Sanctum::ignoreMigrations();
  }
  // ...
}
```

Después, lanzamos `php artisan migrate:fresh` para ejecutar desde cero las migraciones, y esa tabla habrá desaparecido.

3. 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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. 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:

```
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
// ó, si no funciona:
// sudo docker-compose exec myapp 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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
// ó, si no funciona:
// sudo docker-compose exec myapp 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:

```php
// ...
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:

```blade
@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:

```php
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 una de estas líneas en el método `boot` del *provider* `App\Providers\AppServiceProvider`, dependiendo de la versión de Bootstrap que queramos usar (por defecto se usará la 5 en la versión actual de los apuntes):

```php
// Bootstrap 4
Paginator::useBootstrap();
// Bootstrap 5
Paginator::useBootstrapFive();
```

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

```php
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:

```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):

```php
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:

```php
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:

```php
// ...
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.

```php
$libro = Libro::findOrFail($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`.

```php
$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:

```php
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.

```php
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:

```php
<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:

```php
<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.

Ejercicios: realizar la primera parte de ejercicios.

4. relaciones entre modelos

En esta sesión veremos qué tipos de relaciones se pueden establecer entre los modelos de la aplicación, y cómo se reflejan automáticamente en la base de datos. Para ello, Eloquent permite definir relaciones de varios tipos entre tablas. Éstas se definen a través de los distintos modelos involucrados en la relación, como veremos a continuación.

4.1. 1. Relaciones uno a uno o *one to one*

Supongamos que tenemos dos modelos `Usuario` y `Telefono`, de modo que podemos establecer una relación *uno a uno* entre ellos: un usuario tiene un teléfono, y un teléfono pertenece a un usuario.

Para reflejar esta relación en tablas, una de las dos debería tener una referencia a la otra. En este caso, podríamos tener un campo `telefono_id` en la tabla de `usuarios` que indique el teléfono que pertenece a dicho usuario, o viceversa (un campo `usuario_id` en la tabla `telefonos` que indique a qué usuario pertenece un teléfono). Conceptualmente es más correcta la primera opción (el usuario *tiene* el teléfono), así que seguiremos ese primer ejemplo. Es importante que el campo nuevo en la tabla `usuarios` se llame `telefono_id`, como veremos a continuación.

Para indicar que *un usuario tiene un teléfono*, añadimos un nuevo método en el modelo de `Usuario`, que se llame igual que el modelo con el que queremos conectar (`telefono`, en este caso). Dentro, usaremos el método `hasOne` del modelo de usuario para indicar que un objeto de este tipo *tiene un* objeto del otro modelo (teléfono):

```php
class Usuario extends Model
{
  public function telefono()
  {
     return $this->hasOne(Telefono::class);
  }
}
```

Ahora, si queremos obtener el teléfono de un usuario, basta con que hagamos esto:

```php
$telefono = Usuario::findOrFail($id)->telefono;
```

Hemos empleado una característica de Eloquent denominada *propiedades dinámicas*, por la cual podemos referenciar un método de relación como si fuera una propiedad (en lugar de usar `telefono()`, hemos empleado `telefono`).

La instrucción anterior obtiene el objeto `Telefono` asociado con el usuario buscado (a través del `$id` del teléfono). Para que esta asociación tenga efecto, es preciso que en la tabla `usuarios` exista un campo `telefono_id` y que se corresponda con un campo `id` de la tabla de `telefonos`, de modo que Eloquent establece la conexión entre una y otra tabla. Deberemos definir una nueva migración de modificación sobre la tabla *usuarios* para añadir ese nuevo campo, o refrescar la migración original con él y borrar los contenidos previos.

Si queremos utilizar otros campos distintos en una y otra tabla para conectarlas, debemos indicar dos parámetros más al llamar a `hasOne`. Por ejemplo, así relacionaríamos las dos tablas anteriores, indicando que la clave ajena de `usuarios` a `telefonos` es `idTelefono`, y que la clave primaria de `telefonos` a la que se referencia es `codigo`:

```
return $this->hasOne(Telefono::class, 'idTelefono', 'codigo');
```

También es posible obtener la **relación inversa**, es decir, a partir de un teléfono, obtener el usuario al que pertenece. Para ello, añadimos un método `usuario` en el modelo `Telefono` y empleamos el método `belongsTo` para indicar a qué modelo se asocia:

```
class Telefono extends Model
{
  public function usuario()
  {
     return $this->belongsTo(Usuario::class);
  }
}
```

Nuevamente, podemos especificar otros nombres de clave pasando parámetros adicionales a `belongsTo`, igual que se hace para `hasOne`.

De este modo, si queremos obtener el usuario a partir del teléfono, podemos hacerlo así:

```
$usuario = Telefono::findOrFail($idTelefono)->usuario();
```

4.1.1. Guardar datos relacionados

Supongamos que queremos guardar un usuario con su teléfono asociado. Podemos simplemente guardar el *id* del teléfono como un campo más del usuario:

```
// Buscamos el teléfono que queremos asociar
// (suponiendo que existe previamente)
$telefono = Telefono::findOrFail($idTelefono);
$usuario = new Usuario();
$usuario->nombre = "Pepe";
$usuario->email = "pepe@gmail.com";
$usuario->telefono_id = $telefono->id;
$usuario->save();
```

Pero, además, podemos vincular ambos objetos en la relación, usando el método `associate`, de este modo:

```
// Buscamos el teléfono que queremos asociar
// (suponiendo que existe previamente)
$telefono = Telefono::findOrFail($idTelefono);
$usuario = new Usuario();
$usuario->nombre = "Pepe";
$usuario->email = "pepe@gmail.com";
$usuario->telefono()->associate($telefono);
$usuario->save();
```

4.2. Relaciones uno a muchos o *one to many*

Para ilustrar esta relación veamos otro ejemplo: supongamos que tenemos los modelos `Autor` y `Libro`, de modo que un autor puede tener varios libros, y un libro está asociado a un autor.

La forma de establecer la relación entre ambos consistirá en añadir en la tabla de `libros` una clave ajena al autor al que pertenece. A la hora de plasmar esta relación en los modelos, se hace de forma similar al caso anterior, sólo que en lugar de utilizar el método `hasOne` en la clase `Autor` usaríamos el método `hasMany`:

```
class Autor extends Model
{
  public function libros()
  {
     return $this->hasMany(Libro::class);
  }
}
```

Igual que ocurría antes, se asume que la tabla de libros tiene una clave primaria `id`, y que la clave ajena correspondiente hacia la tabla de autores es `autor_id`. De lo contrario, se pueden especificar otros pasando más parámetros a `hasMany`.

De este modo obtenemos los libros asociados a un autor:

```
$libros = Autor::findOrFail($id)->libros();
```

Finalmente, también podemos establecer la **relación inversa**, y recuperar el autor al que pertenece un determinado libro, definiendo un método en la clase `Libro` que emplee `belongsTo`, como en las relaciones uno a uno:

```
class Libro extends Model
{
  public function autor()
  {
     return $this->belongsTo(Autor::class);
  }
}
```

Y obtener, por ejemplo, el nombre del autor a partir del libro:

```
$nombreAutor = Libro::findOrFail($id)->autor->nombre;
```

4.2.1. Aplicando esta relación en nuestro ejemplo

Esta relación la podemos dejar plasmada en nuestro ejemplo de la biblioteca, definiendo un nuevo modelo `Autor` con su correspondiente migración, y relacionando los modelos. Para ello, seguiremos estos pasos:

- Creamos una nueva migración de modificación sobre la tabla de *libros*, para añadir un nuevo campo `autor_id`.

```
php artisan make:migration nuevo_campo_autor_libros --table=libros
```

```
class NuevoCampoAutorLibros extends Migration
{
  public function up()
  {
     Schema::table('libros', function(Blueprint $table) {
         $table->integer('autor_id');
     });
  }

  public function down()
  {
     Schema::table('libros', function(Blueprint $table) {
         $table->dropColumn('autor_id');
     });
  }
}
```

```
php artisan migrate
```

- Creamos de golpe el modelo, la migración y el controlador de autores (aunque el controlador no lo vamos a utilizar, al menos por el momento). El modelo `Autor` debe quedar en la carpeta `app\Models`, junto con el de usuarios y el de libros.

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

> **NOTA**: en este punto, deberás renombrar a mano la migración, ya que el plural que asignará Laravel por defecto será *autors*, y no *autores*. Recuerda cambiar tanto el nombre del fichero de la migración, como el nombre de la tabla a la que se referencia en los métodos `up` y `down`.

- Editamos la migración para definir los campos que tendrá la nueva tabla de autores, en su método `up`: un nombre y un año de nacimiento (opcional):

```
return new class extends Migration
{
  public function up()
  {
     Schema::create('autores', function(Blueprint $table) {
         $table->id();
         $table->string('nombre');
         $table->integer('nacimiento')->nullable();
         $table->timestamps();
     });
  }
}
php artisan migrate
```

- Añadimos en el modelo `Autor` que la tabla asociada será `autores` (de lo contrario, considera que será *autors*. Además, definimos una relación de uno a muchos con los libros, añadiendo el método siguiente:

```php
class Autor extends Model
{
  protected $table = 'autores';
  // ...

  public function libros()
  {
     return $this->hasMany(Libro::class);
  }
}
```

- Recíprocamente, añadimos al modelo `Libro` este otro método, para poder recuperar un autor a partir de uno de sus libros:

```php
class Libro extends Model
{
  // ...

  public function autor()
  {
     return $this->belongsTo(Autor::class);
  }
}
```

- Utilizando *phpMyAdmin*, creamos a mano un par de autores en la tabla de autores, y los relacionamos con algunos de los libros que haya en la tabla de libros, añadiendo también a mano el *id* de cada autor en la clave ajena correspondiente de los libros. Por ejemplo:

id	nombre	nacimiento	created_at	updated_at
1	Orson Scott Card	1951	*NULL*	*NULL*
2	J.R.R. Tolkien	1892	*NULL*	*NULL*

id	titulo	editorial	precio	created_at	updated_at	autor_id
1	El juego de Ender	Ediciones R	7.95	*NULL*	*NULL*	1
3	El señor de los anillos	Anagrama	11.25	*NULL*	*NULL*	2

- Para probar cómo funcionan las relaciones, vamos primero a crear un nuevo libro asociado al autor 1. Definimos una ruta de prueba en el archivo `routes/web.php` con este código (deberemos incorporar con `use` los modelos de `Autor` y `Libro`):

```php
Route::get('relacionPrueba', function() {
    $autor = Autor::findOrFail(1);
    $libro = new Libro();
    $libro->titulo = "Libro de prueba " . rand();
    $libro->editorial = "Editorial de prueba";
    $libro->precio = 5;
    $libro->autor()->associate($autor);
    $libro->save();

    return redirect()->route('libros.index');
});
```

- Ahora, modificamos la vista `libros/index.blade.php` para que, en el listado, utilice las relaciones entre tablas para mostrar el nombre del autor entre paréntesis junto al título de cada libro:

```php
@forelse($libros as $libro)
    <li><a href="{{ route('libros.show', $libro) }}">
        {{ $libro->titulo }} ({{ $libro->autor->nombre }})
    </a></li>
@empty
    <li>No se encontraron libros</li>
@endforelse
```

- Podemos probar las dos cosas accediendo respectivamente a estas dos URLs (suponiendo que el servidor está escuchando en *localhost* por el puerto 8000):

```
http://localhost:8000/relacionPrueba
http://localhost:8000/libros
```

4.2.2. Acceso eficiente a datos relacionados. *Eager loading*

En el ejemplo anterior, hemos visto cómo, dado un libro, podemos obtener el nombre del autor de este modo en una vista Blade:

```php
{{ $libro->autor->nombre }}
```

Sin embargo, este código provoca una nueva consulta en la base de datos para buscar los datos del autor asociado al libro, con lo que, para un listado de 100 libros, estaremos haciendo 100 consultas adicionales para extraer la información de los respectivos autores.

Para evitar esta sobrecarga, podemos emplear una técnica llamada *eager loading* (que en español podríamos traducir como *carga apresurada* o *impaciente*). Consiste en emplear el método `with` para indicar qué relación queremos dejar pre-cargada en el resultado. Por ejemplo, si indicamos algo así en la función `index` del controlador de libros:

```php
public function index()
{
    $libros = Libro::with('autor')->get();
    return view('libros.index', compact('libros'));
}
```

4.3. Relaciones muchos a muchos o *many to many*

Estas relaciones son más difíciles de plasmar, ya que es necesario contar con una tercera tabla que relacione las dos tablas afectadas. Pero vayamos por partes...

Para ilustrar este caso, supongamos los modelos `Usuario` y `Rol`, de modo que un usuario puede tener varios roles, y un rol puede ser asignado a varios usuarios. Nuevamente, definimos un método en el modelo `Usuario` que utilice el método `belongsToMany` para indicar con qué otro modelo se relaciona:

```
1  class Usuario extends Model
2  {
3    public function roles()
4    {
5       return $this->belongsToMany(Rol::class);
6    }
7  }
```

Así ya podremos acceder a los roles de un usuario:

```
1  $roles = Usuario::findOrFail($id)->roles;
```

En este caso, al otro lado de la relación hacemos lo mismo: definimos un método en el modelo `Rol` que indique con `belongsToMany` que puede pertenecer a varios usuarios:

```
1  class Rol extends Model
2  {
3    public function usuarios()
4    {
5       return $this->belongsToMany(Usuario::class);
6    }
7  }
```

A efectos de automatización, es decir, para que Eloquent establezca los nexos de forma automática, si queremos establecer una relación muchos a muchos entre un modelo `A` y otro `B`, se asume que existirá otra tabla `a_b` (el orden en que se colocan los nombres de las tablas es alfabético), con los campos `a_id` y `b_id`, que relacionen los dos modelos. En nuetro caso, se asumirá que existe una tabla `rol_usuario` con un campo `rol_id` y otro llamado `usuario_id`, que enlacen con los correspondientes `id` de las tablas de usuarios y roles. Si esto no fuera así, podemos pasar más parámetros a `belongsToMany` para indicarlo.

En el caso de las relaciones muchos a muchos, es posible que nos interese acceder a algún dato de esa tabla intermedia que los relaciona. En ese caso, hacemos uso del atributo `pivot`, predefinido, y que apunta a la tabla o modelo intermedio entre los dos relacionados. Por ejemplo, si quisiéramos obtener la fecha de creación de la relación entre un usuario y un rol, podríamos hacer esto:

```php
$roles = Usuario::findOrFail($id)->roles;

for($roles as $rol)
{
  echo $rol->pivot->created_at;
}
```

4.4. Más información

Sobre estas relaciones existen algunas variantes, y formas de personalizar las tablas y campos afectados. Se puede consultar más información en la documentación oficial de Eloquent.

5. seeders y factories

En las pruebas que hemos hecho hasta ahora, para tener datos con que probar la aplicación, nos hemos limitado a añadirlos a mano desde *phpMyAdmin*, o bien desde el código con algunos datos simples como "Título de prueba 1" o cosas similares.

Dado que los datos de inicio son necesarios para probar algunas funcionalidades básicas de la aplicación, como son las búsquedas y filtrados, y dado que los formularios para dar de alta y gestionar estos datos normalmente no se tienen listos hasta etapas más tardías, puede resultar conveniente disponer de algún mecanismo que genere estos datos de prueba al inicio, sin preocuparnos de tocar la base de datos a mano o alterar el código de la aplicación para ello. En este aspecto, los *seeders* y *factories* juegan un papel importante.

5.1. *seeders*

Los *seeders* son clases especiales que permiten sembrar (*seed*) de contenido una aplicación. Para crearlos, utilizamos el comando `php artisan` como sigue:

```
php artisan make:seeder NombreSeeder
```

Esto creará una clase llamada `NombreSeeder` en la carpeta `database/seeds` (hasta Laravel 7) o `database/seeders` (desde Laravel 8). En el método `run` de dicha clase podemos crear los elementos que necesitemos añadir a la base de datos.

Por ejemplo, vamos a crear en nuestro proyecto *biblioteca* un seeder llamado `LibrosSeeder` para crear libros, y otro llamado `AutoresSeeder` para autores:

```
php artisan make:seeder LibrosSeeder
php artisan make:seeder AutoresSeeder
```

Editamos el método `run` del *seeder* que hemos creado (el de libros, por ejemplo), y definimos este código para crear un autor con un libro asociado (deberemos incorporar con `use` los modelos de `Autor` y `Libro` previamente):

```php
public function run()
{
    $autor = new Autor();
    $autor->nombre = "Juan Seeder";
    $autor->nacimiento = 1960;
    $autor->save();
    $libro = new Libro();
    $libro->titulo = "El libro del Seeder";
    $libro->editorial = "Seeder S.A.";
    $libro->precio = 10;
    $libro->autor()->associate($autor);
    $libro->save();
}
```

De forma similar, podríamos editar el método `run` de `AutoresSeeder` para crear, en este caso, un autor (sin libro asociado, normalmente).

```php
public function run()
{
    $autor = new Autor();
    $autor->nombre = "Autor Suelto";
    $autor->nacimiento = 1951;
    $autor->save();
}
```

5.1.1. añadiendo los *seeders* a la aplicación

Por defecto, los *seeders* que creamos no forman parte de la aplicación aún, en el sentido de que aún no los podemos ejecutar. Para ello, debemos darlos de alta en el *seeder* general, llamado `DatabaseSeeder`, ubicado en la misma carpeta que los *seeders* que definimos:

```php
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        ...
        $this->call(AutoresSeeder::class);
        $this->call(LibrosSeeder::class);
    }
}
```

> **NOTA:** es importante el orden en que colocamos los *seeders* dentro de este método, como veremos después, ya que esto indicará en qué orden se crean los datos de prueba en la aplicación.

5.1.2. lanzar los *seeders*

Si queremos ejecutar los *seeders* para que añadan los datos, emplearemos este comando:

```
php artisan db:seed
```

Esto lanzará todos los *seeders* que tengamos declarados en la clase `DatabaseSeeder`. Si sólo queremos lanzar uno en concreto, podemos hacer lo siguiente:

```
php artisan db:seed --class=LibrosSeeder
```

También puede ser necesario (y a veces conveniente) limpiar la base de datos y llenarla desde cero con los datos de los seeds para empezar a probar la aplicación. En este caso, el comando es el siguiente:

```
php artisan migrate:fresh --seed
```

5.2. *factories*

Los *seeders* son una herramienta útil para poblar nuestra aplicación con datos al inicio. Podemos, por ejemplo, dar de alta una serie de usuarios iniciales con acceso a la aplicación, para que con ellos se puedan rellenar el resto de datos. También podemos dar de alta una serie de datos predefinidos en ciertas tablas, o datos de prueba que luego poder borrar.

Sin embargo, los *seeders* por sí solos se quedan algo "cojos". ¿Qué tendríamos que hacer para dar de alta 10 o 20 libros en nuestra base de datos de *biblioteca*? Tendríamos que definir algún tipo de bucle en el *seeder*, y definir datos diferentes (por ejemplo, con identificadores o contadores aleatorios) para cada libro. Para facilitar esta tarea, podemos echar mano de los *factories*.

Los *factories* son clases que permiten generar datos por lotes. Se crean con el siguiente comando, almacenándose la clase en la carpeta `database/factories`:

```
php artisan make:factory NombreFactory --model=NombreModelo
```

5.2.1. generar *factories* y asociarlos a su modelo

Si recordamos, cuando creamos los modelos de `Autor` y `Libro` la clase se crea con una cláusula `use` que alude al *trait* `HasFactory`.

```
class Libro extends Model
{
    use HasFactory;

    ...
}
```

Un *trait* básicamente es un conjunto de métodos que se puede emplear por cualquier clase que quiera utilizarlos. De este modo, se amortigua en parte la limitación de sólo poder heredar de una clase, y mediante estos *traits* podemos incorporar la funcionalidad de otras. En este caso, indicamos que el modelo de *Libro* puede usar los métodos de su *factory* asociado.

Por ejemplo, vamos a crear un *factory* para generar autores y otro para libros, indicando con el parámetro `--model` el modelo asociado a cada *factory*:

```
php artisan make:factory AutorFactory --model=Autor
php artisan make:factory LibroFactory --model=Libro
```

Esto generará dos clases donde, en una anotación `@extends` se especificará el modelo asociado. Por ejemplo, así quedaría el *factory* de autores:

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Autor>
 */
class AutorFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition()
    {
```

```
        return [
            //
        ];
    }
}
```

En Laravel 8, en lugar de la anotación `@extends`, la relación entre el modelo y el *factory* se daba a través de un atributo protegido llamado `model`. Esta opción sigue estando permitida en Laravel 9:

```
...
class AutorFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = Autor::class;

    ...
}
```

En cualquiera de los dos casos, el método `definition` es el que se va a ejecutar cuando utilicemos el *factory*, para generar los datos del modelo asociado. Podemos devolver datos manualmente generados (por ejemplo, *Autor 1*, *Autor 2*, etc):

```
public function definition()
{
    return [
        'nombre' => "Autor " . rand(1, 100),
        'nacimiento' => rand(1950, 1990)
    ];
}
```

Para crear autores usando este *factory*, lo invocamos desde el *seeder* correspondiente (`AutoresSeeder`, en este caso), en su método `run`:

```
...
class AutoresSeeder extends Seeder
{
    public function run()
    {
        return Autor::factory()->count(5)->create();
    }
}
```

Notar cómo acudimos al *factory* a través del modelo, con `Autor::factory()`, gracias al *trait* `HasFactory`. Esto generará 5 autores con datos distintos y aleatorios.

5.2.2. *fakers*

Estaremos de acuerdo en que generar datos del tipo "Autor 1", "Autor 2", etc, no queda demasiado "real" en una aplicación, por mucho que sean datos de prueba. Por ello, Laravel nos proporciona los *fakers* para generar datos al azar con una apariencia determinada. Así, podemos generar nombres reales aleatorios, o direcciones de correo electrónico, o frases, o textos largos.

Para generar estos datos de prueba, Laravel cuenta con una clase llamada `Faker`, cuyo objeto está automáticamente incorporado en el *factory* a través de la propiedad `$this->faker`. Internamente dispone de métodos para generar datos de distintos tipos.Algunos de los más habituales son:

- `name`: genera un nombre de persona. Admite como parámetro opcional "male" o "female" para generar nombres masculinos o femeninos, respectivamente.
- `sentence`: genera una frase corta. Admite como parámetro opcional un número, indicando cuántas palabras generar.
- `word`: genera una palabra aleatoria.
- `text`: genera un texto largo.
- `phoneNumber`: genera un número de teléfono.
- `email`: genera un e-mail aleatorio.
- `randomNumber`: genera un número aleatorio. Como alternativa, también se tiene `numberBetween`, que genera un número aleatorio entre un mínimo y un máximo pasados como parámetro.
- ... etc (aquí podéis consultar más posibilidades al respecto).

Además, también tenemos disponible el método `unique()` para asegurarnos de que alguno de los campos que generemos no se repita entre registros.

Vamos a generar los datos de autores usando este *faker* en el método `definition` de nuestra `AutorFactory`:

```php
public function definition()
{
    return [
        'nombre' => $this->faker->name,
        'nacimiento' => $this->faker->numberBetween(1950, 1990)
    ];
}
```

Como podemos intuir, generamos un nombre al azar (de hombre o mujer) y un año de nacimiento entre 1950 y 1990.

Del mismo modo, completamos la información del método `definition` para el *factory* de libros (`LibroFactory`):

```php
public function definition()
{
    return [
        'titulo' => $this->faker->sentence,
        'editorial' => $this->faker->sentence(2),
        'precio' => $this->faker->randomFloat(2, 5, 20)
    ];
}
```

5.3. relacionando los modelos

Finalmente, en los *seeder* correspondientes, podemos utilizar estos *factory* para generar N objetos del modelo asociado. Por ejemplo, para el caso de los autores, generamos 5 autores aleatorios:

```php
class AutoresSeeder extends Seeder
{
    public function run()
    {
        Autor::factory()->count(5)->create();
    }
}
```

En el caso de los libros, recorremos los autores y, para cada uno, le generamos 2 libros al azar:

```php
class LibrosSeeder extends Seeder
{
    public function run()
    {
        $autores = Autor::all();
        $autores->each(function($autor) {
            Libro::factory()->count(2)->create([
                'autor_id' => $autor->id
            ]);
        });
    }
}
```

Recordemos ahora algo que hemos comentado antes en este mismo documento: es IMPORTANTE el orden en que se declaran los *seeders* en la clase `DatabaseSeeder`: si primero colocamos el de libros y luego el de autores, aún no habrá autores con los que generar los libros. Por eso es importante generar primero los autores y así, usarlos para recorrerlos y generar los libros.

6. *query builder* y uso de fechas

En este documento veremos un par de cuestiones secundarias que nos han quedado en el tintero. La primera es una forma alternativa de "atacar" la base de datos, sin emplear el ORM Eloquent: el *query builder*. La segunda es cómo trabajar cómodamente con fechas en aplicaciones Laravel.

6.1. *query builder*

A la hora de obtener datos de la base de datos, en lugar de usar modelos de Eloquent, podemos emplear también el *query builder*, una herramienta incorporada con Laravel que permite realizar operaciones sobre la base de datos sin utilizar un modelo de objetos por detrás, y con una sintaxis diferente a SQL.

6.1.1. consultas

Para utilizar estas consultas, utilizamos el elemento `DB`. Dicho elemento corresponde al espacio de nombres `Illuminate\Support\Facades\DB` que deberemos incluir. Internamente, tiene un método `table` para especificar la tabla sobre la que se quiere consultar. Una vez referenciada, con el método `get` obtenemos todos los registros:

```
1 use Illuminate\Support\Facades\DB;
2 //...
3 $personas = DB::table('personas')->get();
```

A pesar de no estar trabajando con clases, lo que obtenemos aquí es un array de objetos, no un array asociativo.

En el caso de buscar un registro concreto (por su *id*, por ejemplo), utilizamos el método `where`, pasándole como parámetros el nombre del campo a comparar, y el valor que debe tener. Después, enlazamos con el método `first` para obtener sólo el primer registro de la búsqueda (de lo contrario, obtendríamos un array con un resultado, si buscamos por *id*):

```
1 $persona = DB::table('personas')->where('id', $id)->first();
```

6.1.2. actualizaciones

Si lo que queremos hacer es una **inserción**, empleamos el método `insert` de la tabla. En este caso, le pasamos un array asociativo con los nombres de cada campo del nuevo registro, y sus valores:

```
1 DB::table('personas')->insert([
2   'nombre' => 'Juan',
3   'edad' => 56
4 ]);
```

En el caso de **modificaciones**, utilizamos el método `where` para filtrar el registro o registros a modificar, y empleamos el método `update` con el array de campos a modificar:

```php
DB::table('personas')->where('id', $id)->update([
  'nombre' => 'Juan',
  'edad' => 56
]);
```

Para **borrados**, usamos una estructura similar a la anterior, reemplazando la llamada a `update` por `delete`, que no necesita parámetros:

```php
DB::table('personas')->where('id', $id)->delete();
```

6.2. uso de fechas

En algunas tablas que hemos visto o creado, se ha usado un tipo *timestamp*, que básicamente genera un tipo fecha en la tabla correspondiente. Estos campos de tipo tabla son instancias de una librería PHP llamada *Carbon*, muy útil para trabajar con fechas. Así que, si tenemos un registro de tipo `Persona` con un campo `created_at` de tipo fecha, podemos trabajar con él como una fecha *Carbon*, y, por ejemplo, mostrarla en una vista con un formato específico:

```php
<p>
  Fecha creación:
  {{ Carbon\Carbon::parse($persona->created_at)->format('d/m/Y') }}
</p>
```

Además, para trabajar sobre los campos `created_at` y `updated_at` que por defecto se crean en una tabla desde una migración Laravel, podemos emplear esta librería *Carbon* para darles valor, aunque de esto ya se encarga Eloquent automáticamente, pero por si lo queremos hacer manualmente, aquí va un ejemplo:

```php
DB::table('personas')->insert([
  'nombre' => 'Juan',
  'edad' => 56,
  'created_at' => Carbon::now(),
  'updated_at' => Carbon::now()
]);
```

Para poder emplear la clase `Carbon`, debemos importarla (`use Carbon\Carbon`), o bien anteponerle siempre el prefijo del *namespace* `Carbon\Carbon`, como en el ejemplo de `format` anterior.

7. bibliografia

- Nacho Iborra Baeza.