Sesión 5. El modelo y los datos (II): *seeders*, *factories* y relaciones entre modelos

En esta sesión continuaremos con lo visto en la anterior en cuanto a mecanismos de acceso a datos desde Laravel, y hablaremos de conceptos algo más avanzados. Por un lado, veremos cómo podemos poblar las tablas de nuestra base de datos con una serie de datos ya pre-cargados, e incluso con datos ficticios que nos sirvan para unas pruebas iniciales, que luego se puedan descartar. También veremos qué tipos de relaciones se pueden establecer entre los modelos de la aplicación, y cómo se reflejan automáticamente en la base de datos.

1. Relaciones entre modelos

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.

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 `usuario_id` en la tabla de `telefonos` que indique a quién pertenece dicho teléfono. Es importante que el campo se llame `usuario_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):

```
class Usuario extends Model
{
    public function telefono()
    {
        return $this->hasOne('App\Models\Telefono');
    }
}
```

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

```
$telefono = Usuario::findOrFail($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 usuario). Para que esta asociación tenga efecto, es preciso que en la tabla `telefonos` exista un campo `usuario_id` y que se corresponda con un campo `id` de la tabla de `usuarios`, de modo que Eloquent establece la conexión entre una y otra tabla. Deberemos definir una nueva migración de modificación sobre la tabla *telefonos* 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 `telefonos` es `idUsuario`, y que la clave local a la que se referencia en `usuarios` es `codigo`:

```
return $this->hasOne('App\Models\Telefono', 'idUsuario', '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 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('App\Models\Usuario');
    }
}
```

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();
```

1.1.1. 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();
```

1.2. 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('App\Models\Libro');
    }
}
```

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('App\Models\Autor');
    }
}
```

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

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

1.2.1. 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:

1. 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');
        });
    }
}
```

```
php artisan migrate
```

2. 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). Moveremos el modelo `Autor` a 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 clase interna, como el nombre de la tabla a la que se referencia en los métodos `up` y `down`.

3. 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):

```
class CreateAutoresTable 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
```

4. 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:

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

    public function libros()
    {
        return $this->hasMany('App\Models\Libro');
    }
}
```

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

```
class Libro extends Model
{
    ...

    public function autor()
    {
        return $this->belongsTo('App\Models\Autor');
    }
}
```

6. Utilizando *phpMyAdmin*, cremos 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

7. 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`):

```
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');
});
```

8. 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:

```
@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
```

9. 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
```

1.2.2. 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:

```
{{ $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:

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

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

1.3. 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 relaciones 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:

```
class Usuario extends Model
{
    public function roles()
    {
        return $this->belongsToMany('App\Models\Rol');
    }
}
```

Así ya podremos acceder a los roles de un usuario:

```
$roles = Usuario::findOrFail($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:

```
class Rol extends Model
{
    public function usuarios()
    {
        return $this->belongsToMany('App\Models\Usuario');
    }
}
```

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:

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

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

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.

2. *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.

2.1. Los *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`:

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

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

```
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();
}
```

2.1.1. 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:

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

2.1.2. Lanzar los *seeders*

Si sólo queremos ejecutar este *seeder* para que añada 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
```

2.2. Los *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
```

Por ejemplo, vamos a crear un *factory* para generar autores:

```
php artisan make:factory AutorFactory
```

2.2.1. Usando *factories* hasta Laravel 8

Hasta la aparición de Laravel 8, los *factories* eran básicamente un archivo PHP en la carpeta anteriormente citada `database/factories`, con un método `define` que debíamos completar con los datos que se van a emplear para generar objetos de esa factoría. Por ejemplo, así generamos autores con nombres al azar ("Autor X") y nacimientos al azar entre 1950 y 1990:

```
use App\Models\Autor;
use Faker\Generator as Faker;

$factory->define(Autor::class, function(Faker $faker) {
    return [
        'nombre' => "Autor " . rand(1, 100),
        'nacimiento' => rand(1950, 1990)
    ];
})
```

Notar también que utilizamos el modelo `Autor` (`Autor::class`) en lugar del modelo que viene por defecto al crear el *factory* (`Model:class`), deberemos cambiarlo en el código.

Ahora, para crear, por ejemplo, 5 autores aleatorios usando este *factory*, creamos el *seeder* correspondiente...

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

... llamamos al *factory* en su método `run` para crear 5 autores...

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

... y damos de alta el nuevo *seeder* en `DatabaseSeeder`:

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

Usando los *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. Si nos fijamos, cuando definimos un *factory* existe un parámetro de tipo `Faker` en la función `define`, que podemos emplear:

```
$factory->define(Autor::class, function(Faker $faker) {
    ...
```

Este objeto `Faker` dispone de una serie de propiedades que generan datos de un cierto tipo. 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.

Volviendo a nuestro ejemplo, vamos a modificar el *factory* de autores para que genere nombres reales, y años de nacimiento entre 1950 y 1990 usando el `Faker`:

```
use App\Models\Autor;
use Faker\Generator as Faker;

$factory->define(Autor::class, function(Faker $faker) {
    return [
        'nombre' => $faker->name,
        'nacimiento' => $faker->numberBetween(1950, 1990)
    ];
})
```

Si ahora actualizamos la base de datos, veremos los nuevos nombres generados:

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

Generando datos relacionados

Para terminar con esta sección, hagamos las cosas bien. Hemos generado autores, pero esos autores escriben libros. ¿Cómo podemos generar N autores, cada uno con M libros asignados?

En primer lugar, vamos a modificar el *factory* de los libros para que genere un título, editorial y precio al azar (el precio entre 5 y 20 euros, por ejemplo, con 2 decimales):

```
use App\Models\Libro;
use Faker\Generator as Faker;

define(Libro::class, function(Faker $faker) {
    return [
        'titulo' => $faker->sentence,
        'editorial' => $faker->sentence(2),
        'precio' => $faker->randomFloat(2, 5, 20)
    ];
})
```

Así generamos 2 libros asignados a cada uno de los 5 autores creados con el *seeder* de autores:

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

Como vemos, lo que hacemos es recorrer los autores previamente creados (por lo que el *seeder* de autores debe lanzarse ANTES que el de libros), y para cada uno, crear 2 libros asociados a ese *id* de autor.`

2.2.2. Usando *factories* desde Laravel 8

Uno de los cambios importantes que ha traído la versión 8 de Laravel es que ahora los *factories* están orientados a objetos, por lo que se engloban en clases. Además, por defecto se asocian a los modelos que

creamos, de forma que podemos generar una factoría de objetos a partir de una clase, como veremos a continuación. Por este motivo, cuando creamos un modelo se añade una cláusula `use` indicando que emplea el *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.

Cuando creamos una factoría en Laravel 8 empleando el comando `php artisan make:factory` comentado anteriormente, obtendremos una clase con el nombre que hayamos indicado, en la carpeta `database/factories`. Por ejemplo:

```
namespace Database\Factories;

use App\Models\Autor;
use Illuminate\Database\Eloquent\Factories\Factory;

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

    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            //
        ];
    }
}
```

Ahora deberemos rellenar el método `definition` con los datos que queramos generar para cada objeto que se cree. Por ejemplo, así emplearíamos el `faker` (ahora automáticamente incorporado en el propio objeto `$this`), para generar datos al azar para los autores:

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

Finalmente, en el *seeder* correspondiente, podemos utilizar este *factory* para generar N objetos del modelo asociado. Por ejemplo:

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

Para generar datos relacionados entre modelos (por ejemplo, libros con sus autores), procedemos igual que en las versiones anteriores de Laravel, pero teniendo en cuenta que para llamar a la factoría se debe utilizar el método estático del modelo asociado. Por ejemplo:

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

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

3. Más opciones: *query builder* y uso de fechas

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.

3.1. Consultas

Para utilizar estas consultas, utilizamos el elemento `DB`. Podemos emplearlo directamente anteponiéndole una barra invertida, o incluir con `use` la clase (simplemente `use DB`). Este elemento 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:

```
$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*):

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

3.2. 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:

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

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:

```
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:

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

3.3. 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:

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

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:

```
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.

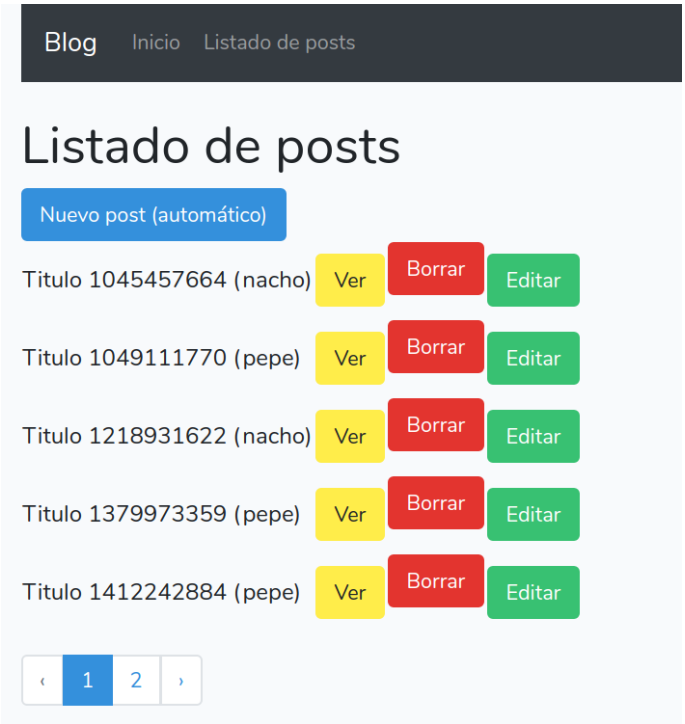
En este punto, puedes realizar el Ejercicio 3 del final de la sesión, que es de carácter optativo.

4. Ejercicios propuestos

Ejercicio 1

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

- Crea una relación *uno a muchos* entre el modelo de `Usuario` y el modelo de `Post`, ambos ya existentes en la aplicación, de forma que un post es de un usuario, y un usuario puede tener muchos posts. Deberás definir una nueva migración de modificación sobre la tabla *posts* que añada un nuevo campo `usuario_id`, y establecer a partir de él la relación, como hemos hecho en el ejemplo con autores y libros.
- Crea desde *phpMyAdmin* una serie de usuarios de prueba en la tabla *usuarios*, y asocia algunos de ellos a los posts que haya.
- Modifica la vista `posts/index.blade.php` para que, junto al título de cada post, entre paréntesis, aparezca el login del usuario que lo creó.



Ejercicio 2

Continuamos con el proyecto **blog** anterior. Ahora añadiremos lo siguiente:

- Crea un *seeder* llamado `UsuariosSeeder`, con un factory asociado llamado `UsuarioFactory` (renombra el que viene por defecto `UserFactory` para aprovecharlo). Crea con esto 3 usuarios de prueba, con *logins* que sean únicos y de una sola palabra (usa el *faker*), y passwords también de una sola palabra, sin encriptar (para poderlos identificar después, llegado el caso).
- Crea otro *seeder* llamado `PostsSeeder` con un factory asociado llamado `PostFactory`. En el *factory*, define con el *faker* títulos aleatorios (frases) y contenidos aleatorios (textos largos). Usa el *seeder* para crear 3 posts para cada uno de los usuarios existentes.

Utiliza la opción `php artisan migrate:fresh --seed` para borrar todo contenido previo y poblar la base de datos con estos nuevos elementos. Comprueba después desde la página del listado de posts, y

desde *phpMyAdmin*, que la información que aparece es correcta.

Ejercicio 3

Opcional

Añade al proyecto **blog** un nuevo modelo llamado `Comentario`, junto con su migración y controlador asociados. Cada comentario tendrá como campo el contenido del comentario, y estará relacionado *uno a muchos* con el modelo `Usuario`, de forma que un usuario puede tener muchos comentarios, y cada comentario pertenece a un usuario. También tendrá una relación *uno a muchos* con el modelo `Post`, de modo que un comentario pertenece a un post, y un post puede tener muchos comentarios. Por lo tanto, la migración de los comentarios deberá tener como campos adicionales la relación con el usuario (`usuario_id`) y con el post al que pertenece (`post_id`).

Aplica la migración para reflejar la nueva tabla en la base de datos, y utiliza un *seeder* y un *factory* para crear 3 comentarios en cada post, con el usuario que sea. A la hora de aplicar todo esto, borra los contenidos previos de la base de datos (`migrate:fresh --seed`).

> **AYUDA**: si quieres elegir un usuario al azar como autor de cada comentario, puedes hacer algo así:

```
Usuario::inRandomOrder()->first();
```

> En este caso, sería conveniente que ese usuario aleatorio se añada directamente en el *factory* del comentario, y no en el *seeder*, ya que de lo contrario es posible que genere el mismo usuario para todos los comentarios de un post.

En la ficha de los posts (vista `posts/show.blade.php`), añade el código necesario para mostrar el *login* del usuario que ha hecho el *post*, y el listado de comentarios asociado al post, mostrando para cada uno el *login* del usuario que lo hizo, y el texto del comentario en sí. Utiliza también la librería *Carbon* para mostrar la fecha de creación del post (o la de los comentarios, como prefieras) en formato *d/m/Y*.

Aquí tienes una captura de pantalla de cómo podría quedar:

¿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_05.zip`.