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.

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` 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):

```
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).

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

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(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.

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
```

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 --model=NombreModelo
```

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.

2.2. 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.

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`:

```
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`):

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

2.3. 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:

```
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:

```
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.