Unit 6. The model and data (II): *seeders*, *factories* and relationships between models

In this unit we will continue with what we saw in the previous one in terms of data access mechanisms from Laravel, and we will talk about somewhat more advanced concepts. On the one hand, we will see how we can fill the tables of our database with a series of data already pre-loaded, and even with fictitious data that serve us for initial tests, which can then be discarded. We will also look at what types of relationships can be established between the models of the application, and how they are automatically reflected in the database.

1. Relationships between models

Eloquent allows you to define relationships of various types between tables. These are defined through the different models involved in the relationship, as we will see below.

1.1. One-to-one relationships

Suppose we have two models 'User' and 'Phone', so that we can establish a *one-to-one* relationship between them: a user has a phone , and a phone belongs to a user.

To reflect this relationship in tables, one of the two should have a reference to the other. In this case, we could have a field 'user_id' in the table of 'phones' that indicates who owns that phone. It is important that the field is called 'user_id', as we will see below.

To indicate that *a user has a phone*, we add a new method in the 'User' model, which is called the same as the model with which we want to connect ('phone', in this case):

```
class User extends Model

{

    public function phone()

    {

        return $this->hasOne('App\Models\Phone');

    }

}
```

Now, if we want to get a user's phone, just do this:

```
$phone = User::findOrFail($id)->phone;
```

We have used an Eloquent feature called *dynamic properties*, whereby we can reference a relationship method as if it were a property (instead of using 'phone()', we have used 'phone').

The above instruction gets the 'Phone' object associated with the searched user (via the user's '$id'). For this association to take effect, it is necessary that in the 'phones' table there is a field 'user_id' and that it corresponds to a field 'id' of the table of 'users', so that Eloquent establishes the connection between one table and another. We will have to define a new modification migration on the table *phones* to add that new field, or refresh the original migration with it and delete the previous contents.

If we want to use other different fields in one and another table to connect them, we must indicate two more parameters when calling 'hasOne'. For example, this way we would relate the two previous tables, indicating that the foreign key of 'phones' is 'idUser', and that the local key referred to in 'users' is 'code':

```
return $this->hasOne('App\Models\Phone', 'idUser', 'code');
```

It is also possible to obtain the **inverse relationship**, that is, from a phone, to obtain the user to which it belongs. To do this, we add a method in the 'Phone' model and use the 'belongsTo' method to indicate which model it is associated with:

```
class Phone extends Model

{

    public function user()

    {

        return $this->belongsTo('App\Models\User');

    }

}
```

Again, we can specify other key names by passing additional parameters to 'belongsTo', just as is done for 'hasOne'.

In this way, if we want to obtain the user from the phone, we can do it like this:

```
$user = Phone::findOrFail($idPhone)->user();
```

```
```

1.1.1. Save related data

Suppose we want to save a user with its associated phone. We can simply save the * id* of the phone as another user field:

```
We are looking for the phone we want to associate

(assuming it exists previously)

$phone = Phone::findOrFail($idPhone);

$user = new User();

$user->name =  "Pepe";

$user->email = "pepe@gmail.com";

$user->phone_id = $phone->id;

$user->save();
```

But, in addition, we can link both objects in the relationship, using the 'associate' method, in this way:

```
We are looking for the phone we want to associate

(assuming it exists previously)

$phone = Phone::findOrFail($idPhone);

$user = new User();

$user->name =  "Pepe";

$user->email = "pepe@gmail.com";

$user->phone()->associate($phone);

$user->save();
```

1.2. One-to-many relationships

To illustrate this relationship let's look at another example: suppose we have the models 'Author' and 'Book', so that an author can have several books, and a book is associated with an author.

The way to establish the relationship between the two will be to add in the table of 'books' a foreign key to the author to which it belongs. When it comes to translating this relationship into the models, it is done in a similar way to the previous case, only instead of using the 'hasOne' method in the 'Author' class we would use the 'hasMany' method:

```
class Author extends Model

{

    public function books()

    {

        return $this->hasMany('App\Models\Book');

    }

}
```

As before, it is assumed that the book table has a primary key 'id', and that the corresponding foreign key to the authors table is 'author_id'. Otherwise, others can be specified by passing more parameters to 'hasMany'.

In this way we obtain the books associated with an author:

```
$books = Author::findOrFail($id)->books();
```

Finally, we can also establish the **inverse relationship**, and recover the author to which a certain book belongs, defining a method in the ' Book' class that uses 'belongsTo', as in one-to-one relationships:

```
class Book extends Model

{

    public function author()

    {
        return $this->belongsTo('App\Models\Author');

    }

}
```

And obtain, for example, the name of the author from the book:

```
$nameAuthor = Book::findOrFail($id)->author->name;
```

1.2.1. Applying this relationship in our example

This relationship can be reflected in our example of the library, defining a new model 'Author' with its corresponding migration, and relating the models. To do this, we will follow these steps:

1. Create a new modification migration on the *books* table, to add a new 'author_id' field.

```
php artisan make:migration new_field_author_books --table=books
```

```
class NewFieldAuthorBooks extends Migration

{

    public function up()

    {

        Schema::table('books', function(Blueprint $table) {

            $table->integer('author_id');

        });

    }

}
```

```
php artisan migrate
```

2. We created the model, the migration and the author handler at once (although we are not going to use the driver, at least for the moment).

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

3. We edit the migration to define the fields that the new author table will have, in its 'up' method: a name and a year of birth (optional):

```
class CreateAuthorsTable extends Migration
{
    public function up()
    {
        Schema::create('authors', function(Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->integer('born')->nullable();
            $table->timestamps();
        });
    }
}
```

```
php artisan migrate
```

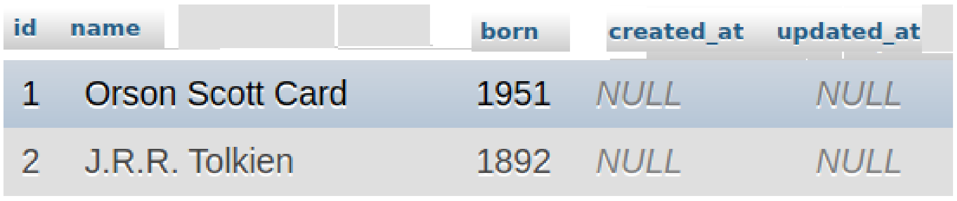
4. We add in the 'Author' model that the associated table will be 'authors' (It's not necessary). In addition, we define a one-to-many relationship with books, adding the following method:

```
class Author extends Model

{

    protected $table = 'authors';



    public function books()

    {

        return $this->hasMany('App\Models\Book');

    }

}
```

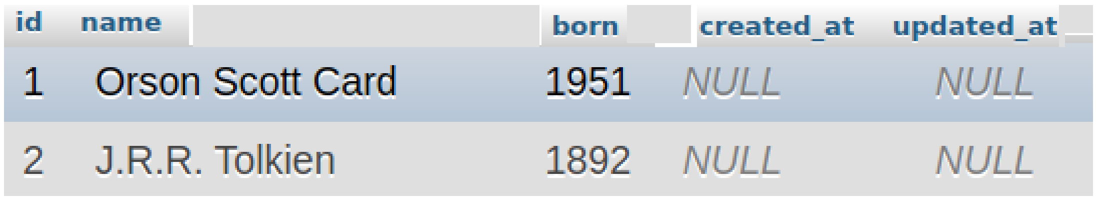
5. Reciprocally, we add to the 'Book' model this other method, to be able to recover an author from one of his books:

```
class Book extends Model

{

    ...



    public function author()

    {

        return $this->belongsTo('App\Models\Author');

    }

}
```

6. Using *phpMyAdmin*, we create by hand a couple of authors in the author table, and relate them to some of the books that are in the book table, also adding by hand the *id* of each author in the corresponding foreign key of the books. For example:



id	name	born	created_at	updated_at
1	Orson Scott Card	1951	NULL	NULL
2	J.R.R. Tolkien	1892	NULL	NULL



id	name	born	created_at	updated_at
1	Orson Scott Card	1951	NULL	NULL
2	J.R.R. Tolkien	1892	NULL	NULL

7. To test how relationships work, let's first create a new book associated with author 1. We define a test path in the 'routes/web.php' file with this code (we must incorporate with 'use' the models of 'Author' and 'Book'):

```php
use App\Models\Book;
use App\Models\Author;

Route::get('relationTest', function() {

    $author = Author::findOrFail(1);

    $book = new Book();

$book->title = "Test Book".  rand();

    $book->editorial = "Test publisher";

    $book->price = 5;

    $book->author()->associate($author);

    $book->save();



    return redirect()->route('books.index');

});
```

8. Now, we modify the 'books/index.blade.php' view so that, in the list, it uses the relationships between tables to display the author's name in parentheses next to the title of each book:

```
@extends('template')
@section('title', 'List of books')
@section('content')
<h1>List of books (database)</h1>
<ul>

@forelse ($books as $book)
<li><a href="{{ route('books.show', $book) }}">{{ $book->title }}
</a>
({{ $book->author->name }})
    <form action="{{ route('books.destroy', $book) }}" method="POST">
        @method('DELETE')
        @csrf
        <button>Delete</button>
       </form>
</li>
@empty
    <li>No books found</li>
@endforelse
</ul>
{{ $books->links() }}
@endsection
```

9. We can test both by respectively accessing these two URLs (assuming the server is listening on *localhost* on port 8000):

```
http://localhost:8010/relationTest

http://localhost:8010/books
```

1.2.2. Efficient access to related data. *Eager loading*

In the example above, we have seen how, given a book, we can get the author's name like this in a Blade view:

```
{{ $book->author->name }}
```

However, this code causes a new query in the database to search for the data of the author associated with the book, so, for a list of 100 books, we will be making 100 additional queries to extract the information of the respective authors.

To avoid this overload, we can use a technique called *eager loading* (which in Spanish we could translate as *carga apresurada* or *impaciente*). It consists of using the 'with' method to indicate what relationship we want to leave pre-loaded in the result. For example, if we indicate something like this in the 'index' function of BookController:

```php
public function index()
{
    $books = Book::with('author')->get();
    return view('books.index', compact('books'));
}
```

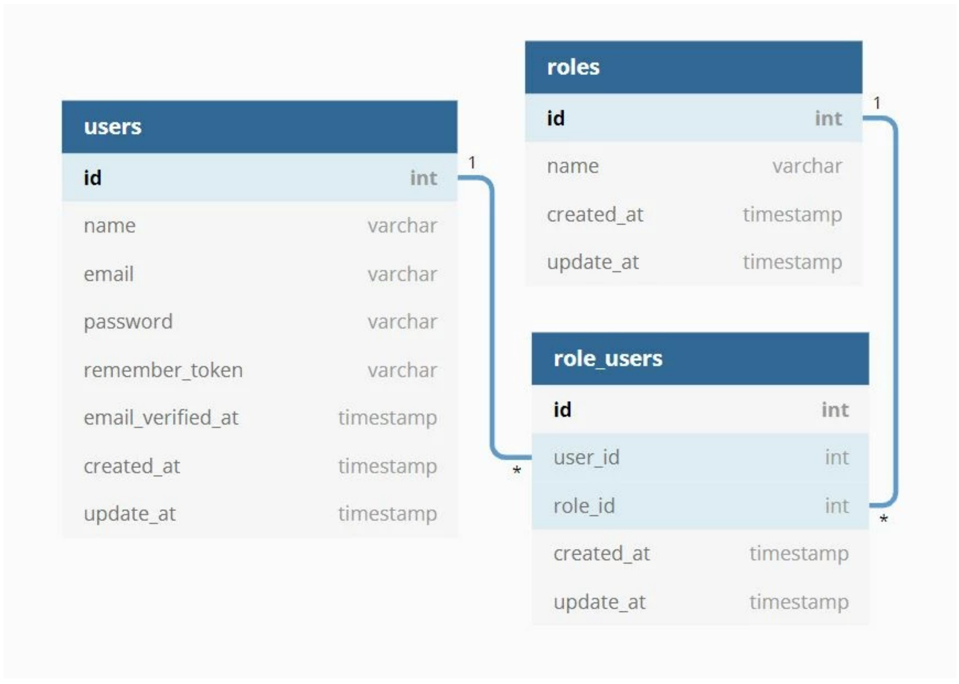
With paginate and orderby:

```php
public function index()
{
    $books = Book::with('author')->orderByDesc('title')->paginate(3);
    return view('books.index', compact('books'));
}
```

1.3. Relationships many to many

These relationships are more difficult to capture, since it is necessary to have a third table that relates the two affected tables. But let's go in parts...

To illustrate this case, suppose the 'User' and 'Role' models, so that a user can have multiple roles, and one role can be assigned to multiple users.



Again, we define a method in the 'User' model that uses the 'belongsToMany' method to indicate which other model it relates to:

```
class User extends Model

{

    public function roles()

    {

        return $this->belongsToMany('App\Models\Role');

    }

}
```

So we can access the roles of a user:

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

In this case, on the other side of the relationship we do the same: we define a method in the 'Role' model that indicates with 'belongsToMany' that it can belong to several users:

```
class Role extends Model

{

    public function users()

    {

        return $this->belongsToMany('App\Models\User');

    }

}
```

For automation purposes, that is, for Eloquent to establish the links automatically, if we want to establish a many-to-many relationship between a model 'A' and another 'B', it is assumed that there will be another table 'a_bs' (the order in which the names of the tables are placed is alphabetical), with the fields 'a_id' and 'b_id', that relate the two models. In our case, it will be assumed that there is a table 'role_users' with a field 'role_id' and another called 'user_id', which link to the corresponding 'id' of the tables of users and roles. If this is not the case, we can pass more parameters to 'belongsToMany' to indicate it.

In the case of many-to-many relationships, we may be interested in accessing some data from that intermediate table that relates them. In that case, we make use of the attribute 'pivot', predefined, and that points to the table or intermediate model between the two related. For example, if we wanted to get the creation date of the relationship between a user and a role, we could do this:

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



for($roles as $role)

{

    echo $role->pivot->created_at;

}
```

On these relationships there are some variants, and ways to customize the affected tables and fields. More information can be found in the official documentation of Eloquent.

2. *Seeders* and *factories*

In the tests we have done so far, to have data with which to test the application, we have limited ourselves to adding them by hand from * phpMyAdmin*, or from the code with some simple data such as "Test title 1" or similar things.

Since the startup data is necessary to test some basic functionalities of the application, such as searches and filtering, and since the forms to register and manage this data are usually not ready until later units, it may be convenient to have some mechanism that generates this test data at startup, without worrying about touching the database by hand or altering the application code for it. In this aspect, seeders and factories play an important role.

2.1. *seeders*

seeders are special classes that allow you to seed (*sembrar*) content in an application. To create them, we use the command 'php artisan' as follows:

```
php artisan make:seeder SeedName
```

This will create a class called 'SeedName' in the 'database/seeders' (from Laravel 8). In the 'run' method of this class we can create the elements that we need to add to the database.

For example, we are going to create in our project *library* a seeder called 'BooksSeeder':

```
php artisan make:seeder BooksSeeder
```

We edit the 'run' method of the *seeder* that we have created, and define this code to create an author with an associated book (we must incorporate with 'use' the models of 'Author' and 'Book' previously):

```
use App\Models\Author;
use App\Models\Book;

class BooksSeeder extends Seeder
{

public function run(): void

    $author = new Author();

    $author->name =  "John Seeder";

    $author->born = 1960;

    $author->save();
    $book = new Book();

    $book->title =  "The Book of the Seeder";

    $book->editorial = "Seeder S.A." ;

    $book->price = 10;

    $book->author()->associate($author);

    $book->save();

}
```

2.1.1. Adding seeders to the app

By default, the *seeders* we create are not part of the application yet, because we cannot execute them yet. To do this, we must register them in the general *seeder*, called 'DatabaseSeeder', located in the same folder as the *seeders* that we define:

```
class DatabaseSeeder extends Seeder

{

   public function run(): void
    {

        ...

        $this->call(BooksSeeder::class);
    }

}
```

2.1.2. Launch the * seeders*

If we only want to run this * seeder* to add the data, we will use this command:

```
php artisan db:seed
```

This will launch all the *seeders* that we have declared in the 'DatabaseSeeder' class. If we only want to launch a specific one, we can do the following:

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

It may also be necessary (and sometimes convenient) to clean the database and fill it from scratch with seed data to start testing the application. In this case, the command is as follows:

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

2.2. *factories*

seeders are a useful tool to fill our application with data at startup. We can, for example, register a series of initial users with access to the application, so that they can fill in the rest of the data. We can also register a series of predefined data in certain tables, or test data that we can then delete.

However, the seeders alone are somewhat "useless". What would we have to do to register 10 or 20 books in our *library* database? We would have to define some kind of loop in the *seeder*, and define different data (for example, with identifiers or random counters) for each book. To facilitate this task, we can use the *factories*.

factories are classes that allow you to generate data in groups. They are created with the following command, storing the class in the 'database/factories' folder:

```
php artisan make:factory NameFactory
```

For example, let's create a * factory* to generate authors:

```
php artisan make:factory AuthorFactory
```

2.2.1. Using * factories*

factories are basically a PHP file in the previous mentioned folder 'database/factories', with a 'define' method that we have to complete with the data that will be used to generate objects from that factory.

One of the important changes that version 8 of Laravel has brought is that now the *factories* are object-oriented, so they are included in classes. In addition, by default they are associated with the models we create, so that we can generate an object factory from a class, as we will see below. For this reason, in the model you have to add 'use HasFactory;' indicating that it uses the *trait* 'HasFactory'. And 'use Illuminate\Database\Eloquent\Factories\HasFactory;' outside the class.

```
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Author extends Model
{

    use HasFactory;

    protected $fillable = ['name', 'born'];

    public function books()
     {
           return $this->hasMany('App\Models\Book');
     }

    ...

}
```

A *trait* is basically a set of methods that can be used by any class that wants to use them. In this way, the limitation of only being able to inherit from one class is partly buffered, and through these *traits* we can incorporate the functionality of others.

When we create a factory using the command 'php artisan make:factory' discussed above, we will get a class with the name we have indicated, in the folder 'database/factories'.

```
namespace Database\Factories;

use App\Models\Author;


use Illuminate\Database\Eloquent\Factories\Factory;


class AuthorFactory extends Factory

{
    /**

     * Define the model's default state.
     *

     * @return array

     */

    public function definition(): array

    {

        return [

            //

        ];

    }

}
```

For example, this is how we generate authors with random names ("Author X") and random births between 1950 and 1990:

```
use Illuminate\Database\Eloquent\Factories\Factory;
use App\Models\Author;

class AuthorFactory extends Factory

{

  public function definition(): array

    {

        return [

        'name' => "Author".  rand(1, 100),

        'born' => rand(1950, 1990)

        ];

    }
}
```

Now, to create, for example, 5 random authors using this *factory*, we create the corresponding *seeder*...

```
php artisan make:seeder AuthorsSeeder
```

... We call the *factory* in their 'run' method to create 5 authors...

```
use App\Models\Author;

class AuthorsSeeder extends Seeder

{


    public function run(): void

    {
        $author = Author::factory()->count(5)->create();

    }

}
```

... and we register the new *seeder* in 'DatabaseSeeder':

```
class DatabaseSeeder extends Seeder

{

    public function run()

    {

        ...

        $this->call(AuthorsSeeder::class);

        $this->call(BooksSeeder::class);

    }

}
```

If we now update the database, we will see the new names generated:

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

Using the * fakers*

We will agree that generating data of the type "Author 1", "Author 2", etc., is not too "real" in an application, even if it is test data. Therefore, Laravel provides us with the *fakers* to generate random data with a certain appearance. Thus, we can generate random real names, or email addresses, or phrases, or long texts. If we look, when we define a *factory* there is a definition() function, which we can use:

```
public function definition(): array
    {

    ...
```

We have a 'Fake' object with a series of properties that generate data of a certain type. Some of the most common are:

- 'name ': generates a person name. It supports as an optional parameter "male" or "female" to generate male or female names, respectively.
- 'sentence ': generates a short sentence. It supports as an optional parameter a number, indicating how many words to generate.
- 'word ': generates a random word.
- 'text ': generates a long text.
- ' phoneNumber': generates a phone number.
- ' Email': generates a random e-mail.
- ' randomNumber': generates a random number. Alternatively, you also have 'numberBetween', which generates a random number between a minimum and a maximum passed as a parameter.

In addition, we also have the 'unique()' method available to ensure that any of the fields we generate are not repeated between records.

For example, this is how we generate authors with random names ("Author X") and random births between 1950 and 1990. We must fill in the 'definition' method with the data we want to generate for each object that is created. For example, we would use fake() to generate random data for the authors:

```
public function definition(): array
```

```
{
    return [
        'name' => fake()->name(),
        'born' => fake()->numberBetween(1950, 1990)
    ];
}
```

If we now update the database, we will see the new names generated:

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

Generating related data

To end this section, let's get things right. We have generated authors, but those authors write books. How can we generate N authors, each with M books assigned?

First, let's create and modify the * factory* of the books so that it generates a title, editorial and random price (the price between 5 and 20 euros, for example, with 2 decimal places):

```
php artisan make:factory BookFactory
```

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


define(Book::class, function(Faker $faker) {
    return [
        'title' => $faker->sentence,
        'editorial' => $faker->sentence(2),
        'price' => $faker->randomFloat(2, 5, 20)
    ];

})
```

First comment this line in AuthorsSeeder because we will create the 5 authors from BooksSeeder

```
class AuthorsSeeder extends Seeder

{


    public function run(): void

    {
       // $author = Author::factory()->count(5)->create();

    }

}
```

Then, we generate 2 books assigned to each of the 5 authors from BooksSeeder:

```


use App\Models\Author;
use App\Models\Book;


class BooksSeeder extends Seeder

{

public function run(): void
{
Author::factory()->count(5)->create()->each(function($author) {
$book = Book::factory()->count(2)->create([ 'author_id' => $author->id ]);
});
}
}

}
```

Remember in the Book model add 'use HasFactory' inside and outside the class. Like this:

```php
<?php

namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    use HasFactory;
    protected $fillable = ['title', 'editorial', 'price'];
    public function author()
{ return $this->belongsTo('App\Models\Author');
}
}
```

Refresh ...

php artisan migrate:fresh --seed

And, As we can see in the database, what we do is go through the previously created authors and for each one, create 2 books associated with that *id* of author.'