Unit6: The Model and data (I): Migrations and Simple Models

In this session we will begin to see some important questions about how Laravel manages access to databases, and what mechanisms it offers to synchronize the data of our application with the documents or records of a database, as well as to automatically generate the structure of tables and fields of the database from the model of the application.

1. Database connection parameters

As you may remember from the introduction unit, we configurated in the '.env' file the database connection parameters:

```
DB_CONNECTION=mysql
DB_HOST=127.21.0.2
DB_PORT=3306
DB_DATABASE=DB_myapp
DB_USERNAME=marta
DB_PASSWORD=''
```

Besides, in the 'config/database.php' file, we changed the default database value from sqlite to mysql

```
'default' => env('DB_CONNECTION', 'mysql'),
```

2. Migrations

Migrations are a kind of version control for a database, and allow you to easily create and modify the schema of that database.

2.1. Structure of migrations

By default, Laravel brings predefined migrations, which are located in the 'database/migrations' folder. Each has a file name that begins with the date it was made, followed by a brief description of what it contains (creation of the user table, sessions, etc...). Some of these migrations (especially the creation of the user table) may help us, but with other fields, so we must edit it, as we will see below.

If we examine the content of a migration, all must have two methods:

- 'up'- allows you to add tables, columns or indexes to the database
- 'down'- reverses what was done by the previous method

If we look at the contents of an 'up' method of those that are predefined to create a table, we see that different methods are used to define the data types of each field in the table, such as 'id()' for fields that can contain autoincremental integers, or 'string()' for text type fields. In addition, there are other modifying methods for adding additional properties, such as 'unique()' to indicate unique values (alternate keys), or 'nullable()' to indicate that a field supports nulls. Here is an example of an 'up' method:

```
public function up()
{
    Schema::create('users', function(BluePrint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        ...
        $table->timestamps();
    });
}
```

By default, as we see in the examples provided, schemas are created with an autonumeric *id*, and *timestamps* to indicate the date of creation and modification of each record, and that Laravel manages automatically when we insert or update content, which is very useful.

On this basis, we can add or remove the fields we want. To see the types available for the columns of the table, we can visit the Laravel documentation on migrations, in particular we will look for the subsection *Available Column Types*. Remember, that the type 'string' that we have used in the previous example has a limitation of 255 characters. For larger texts, the type 'text' (approximately 20,000 characters) or 'longText' can be used.

We can specify a primary key with the 'primary' method, to which we can pass either the name of the key field, or an array of key fields, in the case that it is composed. By default, fields of type 'id' are self-established as primary keys.

```
$table->primary(['field1', 'field2']);
```

2.2. Creating migrations

We create migrations with the command:

```
php artisan make:migration name_migration
```

For example:

```
php artisan make:migration create_table_test
```

Note that Laravel already automatically assigns the date of the migration, we only have to specify the descriptive name of it. Also, if Laravel detects the word *create* in the name of the migration, ending in *table*,it means that it is to create a new table. On the other hand, if it detects the word *to* (among others), and at the end the word *table*,it means that an existing table is going to be altered or modified. This is thanks to the 'TableGuesser' class built into Laravel, which detects certain patterns in migration names. The difference between creation and modification is that the 'up' method of the migration will use 'Schema::create' or 'Schema::table' over the table in question, respectively.

In any case, we can also specify an additional parameter in the migration command to indicate if we want to create or modify a table, and in this way we can define the name of the migration in the language we want, and without pattern restrictions. These two migrations create a table (*orders*) and modify another (*users*), respectively:

```
php artisan make:migration create_table_orders --create=orders
php artisan make:migration new_field_user --table=users
```

In the case of the second migration, if, for example, we want to add (or drop) a column with the phone number of the users, it can look like this (both the 'up' and the 'down' method):

```
public function up()
{
    Schema::table('users', function(Blueprint $table) {
    $table->string('phone')->nullable();
    });
}


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

If we want the field in question to be in a specific order, we can use the 'after' method to indicate behind which field we want to put it (in the 'up' method):

```
$table->string('phone')->after('email')->nullable();
```

2.3. Executing and deleting migrations

To execute the migrations (the 'up' method of each one), we launch the following command from the folder of our project (having previously created the database, and modified the access credentials in the '.env' file):

```
php artisan migrate
```

In addition to the affected tables, there will be another 'migrations' table in the database with a history of the migrations carried out. For each, its *id* (autonumeric), the name of the migration, and the batch process number in which it was done (those that share the same number were made at the same time in the same batch) are stored. In this way, those that have already been done will not be done again.

To undo the migrations made (execute the 'down' method of them), we execute:

```
php artisan migrate:rollback
```

This will remove ALL migrations from the last batch existing in the 'migrations' table. If we do not want to undo everything, but to roll back a certain number of migrations within that batch, we execute the previous command with a '--step' parameter, indicating the number of steps or migrations to undo (in chronological order from most recent to oldest):

```
php artisan migrate:rollback --step=2
```

If we do the migration again, the migrations undone from that lot will be restored.

Another command also widely used is 'migrate:refresh'. What it does is delete all the migrations made and relaunch them. It is useful when, while in development, we add new fields to a table and want to redo the tables completely.

```
php artisan migrate:refresh
```

> **NOTE**: The 'migrate:refresh' command is DESTRUCTIVE, removes the contents of the tables, and should only be used in development environments, not production.

2.4. Applying migrations to our example

We are going to put into practice everything seen in this section to our 'library' project.

Edit the migration for the user table ('create_users_table'), since we will use it in later sessions, and then we edit the 'up' method to leave it like this:

```
public function up()
{
    Schema::create('users', function(Blueprint $table) {
        $table->id();
        $table->string('login')->unique();
        $table->string('password');
        $table->timestamps();
    });
}
```

4. Now let's create a new migration to define the structure of the books:

```
php artisan make:migration create_table_books --create=books
```

5. We then edit the content of this migration, specifically the 'up' method to define these fields in the books:

```
public function up()
    {
        Schema::create('books', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->string('editorial')->nullable();
            $table->float('price');
            $table->timestamps();
        });
    }
```

6. We load the migrations with the command:

```
php artisan migrate:refresh
```

After this, we should already see in our database "*DB_myapp*" the two tables created (*users* and *books*),along with the *migrations* table that Laravel creates to manage the migrations made.

3. Data Model Management

Now that we have the table structure created in the database, let's see what mechanisms Laravel offers to access this data easily from the application. We will see how to define the data model associated with each table, and how to manipulate this data using the Eloquent ORM, incorporated with Laravel.

3.1. Create the model

The idea is to create a class for each table that we have in our database, in order to interact with the table through that associated class. To create this model class, we use the 'make:model' option of the 'php artisan' command. We will pass as an additional parameter the name of the class to be created. For example, in the case of our library, we can create the 'Book' model:

```
php artisan make:model Book
```

By convention, models are created with a singular name, starting with a capital letter, and are placed in the 'app\Models' folder. The basic structure of the model is something like this:

```php
<?php


namespace App\Models;


use Illuminate\Database\Eloquent\Model;


class Book extends Model
{


}


?>
```

In our case, we will also use the user model that already exists in the 'app\Models' folder:

```php
<?php


namespace App\Models;


use Illuminate\Database\Eloquent\Model;


class User extends Authenticatable
{
    ...
```

> **NOTE**: From Laravel 8 the location in the folder 'app\Models' is done by default.

Automatically, this model is associated with a table with the same name, but in plural and lowercase, so the previous models would be associated with tables *books* and *users* in the database, respectively. In case we do not want it to be so, we define a '$table' property in the class with the name that we want the associated table to have. For example:

```php
class Book extends Model
{
    protected $table = 'mybooks';
}
```

3.1.1. Other options for creating models

The previous command 'make:model' supports additional parameters, so that you can create both the model and the migration, and even more, the model, migration and the associated controller. Let's look at some examples:

```
php artisan make:model Movie -m
```

The above command creates a 'Movie' model in the 'app\Models' folder and, in addition, creates a migration called 'create_Movies_table' in the 'database/migrations' folder, ready for us to edit the 'up' method and specify the necessary fields.

Note that the migration name adds an "s" to the table name automatically, starting from the singular model.

```
php artisan make:model Movie -mc
```

This other command creates the same as the previous one, and in addition, a driver called 'MovieController' in the folder 'app\Http/Controllers'. This driver is empty, so that we can add the methods we consider.

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

This other option creates the same as the previous one, but the 'MovieController' controller is in this case a resource controller, so it has already incorporated the set of methods of this type of controllers: 'index', 'show', etc.

We can also use the extended version of these parameters. For example:

```
php artisan make:model Movie --migration --controller --resource
```

In our case, as we have been creating the controllers and migrations before the models, it would not be necessary to take this step, but now that we begin to see how everything works and interrelates, it may be useful to use this command to create all the parts involved (model, migration and controller) at once.

3.1.2. Follow a uniform nomenclature

Remember that, from previous sessions, we have commented on the recommendation/need to follow a uniform nomenclature in the models, controllers and views. Thus, for the 'Book' model we would already have its associated controller 'BookController',and the views would be defined in the subfolder 'resources/views/books',with the names corresponding to each method of the controller (for example, 'index.blade.php',or 'show.blade.php').

3.2. Operations on the model. Getting started with Eloquent

Eloquent is the default ORM built into Laravel. An ORM (*Object Relational Mapping*) is a tool that allows you to establish a relationship between the records of a table in the database and the objects of a class (PHP in our case), so that the data in the database is converted to PHP objects and vice versa. In addition, Eloquent implements the *Active Record* pattern, which adds to the classes methods such as 'save', 'update', 'delete'... that allow you to interact with the database to insert, modify or delete records associated with objects, respectively.

3.2.1 Search

Once the model is created, and even if it is empty, we can already use it in the controllers to access the data. Simply import the corresponding class (with 'use'),and use the methods that are inherited from 'Model'. For example, the 'get' method allows you to get the records from the table, converted to objects. Here's how we'd get all the books in the table from a controller:

```
...
use App\Models\Book;
...


class BookController extends Controller
{
    public function index()
    {
        $books =  Book::get();
        return view('books.index', compact('books'));
    }
}
```

What we get is an array of objects, so we must access their properties. For example, if we want to display book titles in a Blade view, we'd do something like this:

```
@forelse($books  as  $book)
{{ $book->title  }}
@empty
   No books found
@endforelse
```

Alternatively, we can also obtain a **filtered query**,specifying with the 'where' method we can specify a condition for the records. For example, this way we would get the books whose price is less than 10 euros:

```
$books =  Book::where('price',  '<',  10)->get();
```

In this other way we would get books with a price of less than 10 euros and more than 5 euros, so that we can combine conditions:

```
$books =  Book::where('price',  '<',  10)
    ->where('precio', '>', 5)->get();
```

On these base queries we can apply a number of additions. For example, we may want to sort the books by title, for which we would do this in the controller:

```
$books =  Book::orderBy('title')->get();
```

The 'orderBy' method supports a second parameter that indicates the direction of the sort. By default it is 'ASC' (ascending), but it can also be 'DESC':

```
$books =  Book::orderBy('title',  'DESC')->get();
```

Results paginations

If we want to paginate the results obtained,when we obtain the list from the controller, indicate with 'paginate' how many records we want per page:

```
public function index()
{
    $books =  Book::paginate(5);
    return view('books.index', compact('books'));
}
```

Then, in the associated view ('books.index' in the previous example), we can use the 'links' method to display the pagination buttons in the desired place:

```
@forelse($books  as  $book)
{{ $book->title  }}
@empty
   No books found
@endforelse



{{ $books->links() }}
```

If we want to sort the list, we can use "orderBy" or "orderByDesc", passing as a parameter the name of the field by which to sort, before pagination. We can even sort by multiple concatenated criteria:

```
public function index()
{
    $books =  Book::orderByAsc('title')
        ->orderByAsc('editorial')
        ->paginate(5);
    return view('books.index', compact('books'));
}
```

Paginations from Laravel 9

From version 9 of Laravel, the style of the paging buttons changed. If we want to continue using bootstrap, we must add this line "Paginator::useBootstrapFive();" in the 'boot' method of the *provider* 'App\Providers\AppServiceProvider':

```
 Paginator::useBootstrapFive();
```

In addition, we must incorporate the 'use' clause to locate the 'Paginator' element, at the begining:

```
use Illuminate\Pagination\Paginator;
```

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Pagination\Paginator;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register(): void
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot(): void
    {
        Paginator::useBootstrapFive();
    }
}
```

3.2.2. Individual object tabs

A fairly common operation is to display a tab of an object from a list, clicking on the title or some visible part of that object. For example, if we want to see the data of a book from a list with its titles, we can do something like this in the Blade template:

```
@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 }}</li>
@empty
    <li>No books found</li>
@endforelse
</ul>
{{ $books->links() }}
@endsection
```

We see that we have used the 'route' method to indicate the route to follow, with a second parameter, which in this case is the specific object of that row. Laravel will automatically replace it in the link with the identifier of that object.

The route associated with this link could be something like this (in the routes file):

```
Route::get('/books/{id}', [BookController::class, 'show'])
->name('books.show');
```

Although we can also have defined the routes as a package of resources, and each one will have its associated method:

```
Route::resource('books', BookController::class);
```

Finally, the 'show' method of the associated controller will be responsible for obtaining the Book data and generating the corresponding view. Thus, we could generate a view with the data like this:

```
...
class BookController extends Controller
{
    ...
public function show(book $book)
    {

        return view('books.show', compact('book'));
    }

}
```

Obviously you have to create the show view inside books folder in views

```
@extends('template')

@section('Title','Book Tab')

@section('content')
<h1>Book title {{ $book->title }}</h1>
<h2>Editorial {{ $book->editorial }} </h2>
<h2>Price {{ $book->price }} </h2>
Created: {{ $book->created_at}}

@endsection
```

At this point, and in the absence of us being able to make insertions later, you can try to insert a few test books in the *DB_myapp* database from phpMyAdmin, and test these two paths that we have done (listing and book options).

3.2.3. Inserts

Inserts through Eloquent can be performed by creating an instance of the object, populating its attributes, and calling the 'save' method, inherited from the 'Model' superclass.

```
$book =  new  Book();
$book->title  =  "Ender's Game";
$book->editorial   =  "B Editions";
$book->price  =  8.95;
$book->save();
```

As an alternative, you can also use the 'create' method of the model, and pass all the data of the request, which would arrive from a form, as we will see later:

```
book::create($request->all());
```

For the latter to work, two premises must be met:

- Each field in the request must have a field of the same name associated with it in the model.
- We must define in the model a property called '$fillable' with the names of the fields of the request that we are interested in processing (the rest are discarded). This is mandatory to specify, even if we are interested in all fields, to avoid malicious insertions (for example, editing the source code to add other fields and modify unexpected data).

```
class Book extends Model
{
protected $fillable  = ['title',  'editorial',  'price'];
}
```

This insertion code (either field by field, or using the 'all' method) is usually put in the 'store' method of the controller, so that it receives the data from the insertion form and does it in the database. Since we don't know how to work with forms by the moment, we will create a new method in the controller to insert 'newBook'.

newBook method:

```
public function newBook()
    {
        $book = new Book();
        $book->title = "Ender's Game";
        $book->editorial = "B Editions";
        $book->price = 8.95;
        $book->save();
    }
```

Create a new route:

```
Route::get('newBook', [BookController::class, 'newBook']);
```

3.2.4. Modifications

The modification consists of two steps:

- Find the object to be modified To obtain the data of an object from its identifier, we can use the 'find' method of the model, passing the identifier as a parameter.

```
$book = Book::find($id);
```

In case that the object is not found (because, for example, we use a wrong id), the generated view will fail. To avoid this, instead of the 'find' method we can use 'findOrFail', which, in case the object is not found, will generate a view with a 404 error, more appropriate. Also, remember that you can customize these error pages by defining the corresponding views

```
$book = Book::findOrFail($id);
 return view('books.show', compact('book'));
```

- Modify the properties that are needed, and call the 'save' method of the object to save the changes.

For example:

```
$booktoModify =  Book::findOrFail($id);
$booktoModify->title="Other title";
$booktoModify->save();
```

We can also use the 'update' method linked to 'findOrFail',and pass as a parameter all the data of the request, as explained for the insertion, and as long as we have declared the attribute '$fillable' in the model to indicate which fields are accepted:

```
Book::findOrFail($id)->update($request->all());
```

This modification code is usually put in the 'update' method of the driver, so that it receives the data from the edit form and makes the corresponding modification. We will see when we start the unit of forms in Laravel. By the moment we will create a new method in the Bookcontroller 'editBook'.

```
public function editBook($id)
    {
    $booktoModify = Book::findOrFail($id);
    $booktoModify->title="Other title";
    $booktoModify->save();
    }
```

3.2.5. Deleted

To delete, we received the object to be deleted as a parameter:

We will do this normally in the 'destroy' method of the Controller in question. Then, we can redirect books.index view to verify that this $book has been deleted. A redirect is actually changing the url which means you actually invoke index controller's method

```
public function destroy(book $book)
{

      $book->delete();
      return redirect()->route('books.index');
}
```

About delete from views

Typically, delete is triggered by clicking on an item in a view. For example, by clicking on a button or link that puts "Delete". However, if we implement this:

```
<a href="{{ route('books.destroy', $book }}">
Delete
</a>
```

If we want to delete the book with *id* 3, a path *http://library/books/3* will be generated. We can check it by passing the mouse through the link and seeing the lower status bar of the browser. This path, however, will send us to the tab of book 3, not to the delete, since we are sending a GET request, and not a delete request.

To avoid this, the delete option must always be done from a form, where through the helper '@method' we indicate that it is a delete request (DELETE). So the "link" to delete a book would look like this:

```
<form action="{{ route('books.destroy', $book) }}" method="POST">
    @method('DELETE')
    @csrf
    <button>Delete</button>
</form>
```

> **NOTE** the helper '@csrf' we will see it in more detail when talking about forms, but it is added to Laravel forms to avoid attacks of type *cross-site*, that is, access to a URL of our website from other websites.

Change the web routes file, to accept all the methods (in order to accept destroy)

```
Route::resource('books', BookController::class)
->only(['index', 'show', 'create', 'edit']);


Delete the only:

Route::resource('books', BookController::class);
```

And include in the index.blade

```
@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 }}
    <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
```