Laravel: Database access & Models – Part 2

DWES

5.- Eloquent ORM

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database.

When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table.

In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

Models typically are stored in the app\Models directory

5.- Eloquent ORM

You may use the make:model Artisan command to generate a new model:

```
php artisan make:model Flight
```

By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the **Flight** model stores records in the **flights** table.

You may use the **--migration** or -m option to generate a database migration :

```
php artisan make:model Flight --migration
```

You may generate various other types of classes when generating a model, such as factories, seeders, policies, controllers, and form requests.

5.- Eloquent ORM

You may manually specify the model's table name by defining a table property on the model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

5.- Eloquent ORM

Eloquent will also assume that each model's corresponding database table has a **primary key** column named **id**. If necessary, you may define a protected $primaryKey property on your model to specify a different column that serves as your model's primary key:

"Composite" primary keys are not supported by Eloquent models

```php
class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

5.- Eloquent ORM

Eloquent assumes that the **primary key** is an incrementing integer value. If you wish to use a **non-incrementing** or a non-numeric primary key you must define a public $incrementing property on your model that is set to false:

```php
<?php

class Flight extends Model
{
    /**
     * Indicates if the model's ID is auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

5.- Eloquent ORM

If your model's **primary key** is **not an integer**, you should define a protected $keyType property on your model. This property should have a value of string:

```php
<?php

class Flight extends Model
{
    /**
     * The data type of the auto-incrementing ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

5.- Eloquent ORM

Timestamps

By default, Eloquent expects **created_at** and **updated_at columns** to exist on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. If you do not want these columns to be automatically managed by Eloquent, you should define a $timestamps property on your model with a value of false:

```php
class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

5.- Eloquent ORM

Relationships

- One to one (hasOne)

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * Get the phone associated with the user.
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

5.- Eloquent ORM

Relationships

- Inverse One to one (belongsTo)

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

5.- Eloquent ORM

Relationships

- One to many (hasMany)

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

5.- Eloquent ORM

Relationships

- Inverse One to many (belongsTo)

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

5.- Eloquent ORM

Relationships

- Many to many (belongsToMany)

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles(): BelongsToMany
    {
        return $this->belongsToMany(Role::class);
    }
}
```

5.- Eloquent ORM

Relationships

- Inverse many to many (belongsToMany)

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class);
    }
}
```

5.- Eloquent ORM

Data Model Usage

- Query data (all())

```
$users = User::all();

foreach( $users as $user ) {
    echo $user->name;
}
```

With this method, an array of results is obtained in which each element of the array is an instance of the User model, therefore, the fields can be accessed as if they were properties of the object

5.- Eloquent ORM

Data Model Usage

- Eloquent also has search methods by id (find()):

```
$user = User::find(1);
echo $user->name;

// the following methods if they fail will throw a catchable exception
$model = User::findOrFail(1);
$model = User::where('votes', '>', 100)->firstOrFail();
```

In addition, it is allowed to use all the methods seen in “Query Builder”

```
// Obtener 10 usuarios con más de 100 votos
$users = User::where('votes', '>', 100)->take(10)->get();

// Obtener el primer usuario con más de 100 votos
$user = User::where('votes', '>', 100)->first();
```

5.- Eloquent ORM

Data Model Usage

- Other methods as (max() min() avg() sum()):

```
$count = User::where('votes', '>', 100)->count();

$price = Orders::max('price’);

$price = Orders::min('price’);

$price = Orders::avg('price’);

$total = User::sum('votes');;
```

5.- Eloquent ORM

Data Model Usage

- Insert & update data (save() saveMany()):

To insert records into a database table that is associated with a model, you must create an instance of said object, assign values to it, and finally use the save method:

```php
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

$post->comments()->save($comment);
```

5.- Eloquent ORM

Data Model Usage

- Delete data (delete()):

To delete a record from the table, you must retrieve that record and then use the delete method

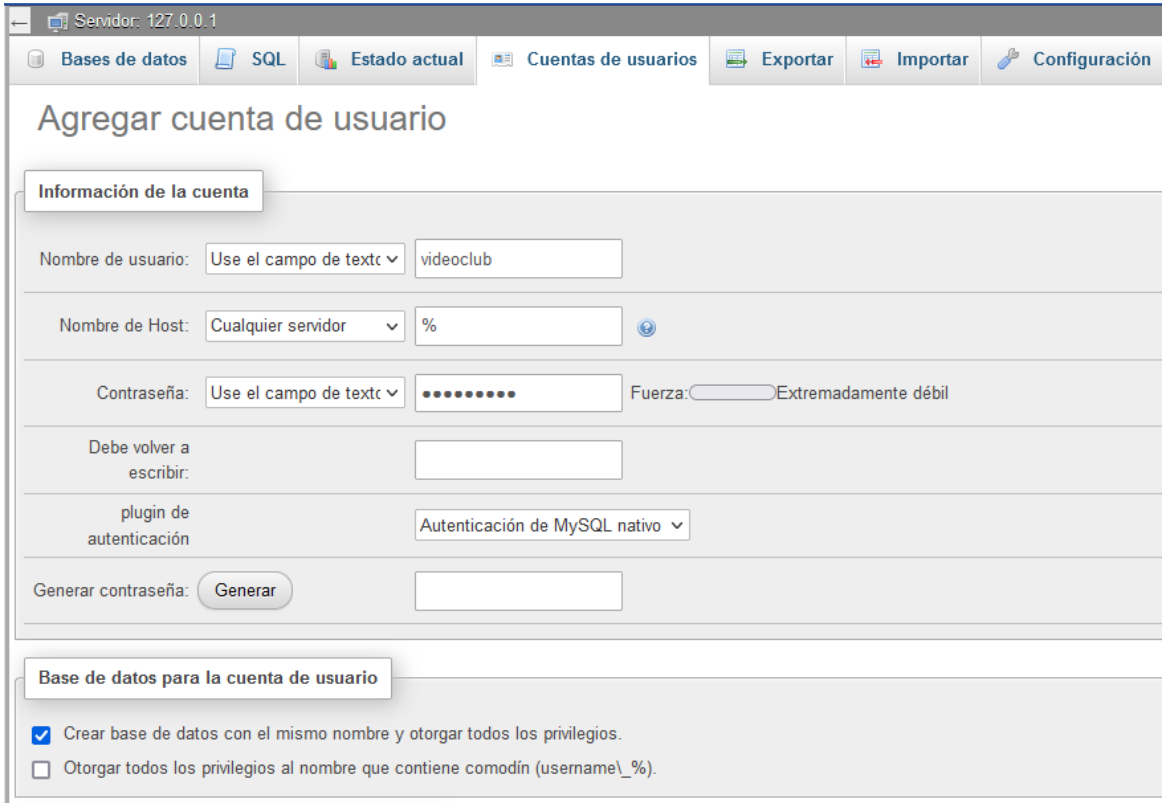
```php
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

Exercise

- Database configuration

Create a user called videoclub in phpMyAdmin, check the option to create a database with the same name.

Configure the project's .env file to indicate that this database and user with will be used in the project



Exercise

- Migration

Since the application only needs one table, we will only create one migration. Using the Artisan command, create the migration for the movies table.

Once the migration is created, the up and down methods must be completed with the necessary fields:

Field	Type	Default value
id	AutoIncrement	
title	String	
year	String of length 8	
director	String of length 64	
poster	String	
rented	Booleano	false
synopsis	Text	
timestamps	Eloquent Timestamps	

Exercise

- Data Model

In order to use the ORM, a Model class must be created for the table that was created previously. Use the Artisan command needed to create the model class that extends Model

Exercise

- Seeds

We are going to create seeds to carry out the tests and have the database filled with minimal data. To do this, you must modify the seed file:
database/seeders/DatabaseSeeder.php

Create a private method in the DatabaseSeeder class called seedCatalog. This method must be called from the run method:

```php
public function run()
{
    // \App\Models\User::factory(10)->create();
    self::seedCatalog();
    $this->command->info('Tabla catálogo inicializada con datos!');
}
```

Exercise

- Seeds

Move the movie array used so far to the DatabaseSeeder seed class. It must be saved as a private attribute of the class. Inside the seedCatalog method, first put the instruction to empty the movies table and then fill the table by going through the array of movies:

```
foreach( $this->arrayPeliculas as $pelicula ) {
    $p = new Movie;
    $p->title = $pelicula['title'];
    $p->year = $pelicula['year'];
    $p->director = $pelicula['director'];
    $p->poster = $pelicula['poster'];
    $p->rented = $pelicula['rented'];
    $p->synopsis = $pelicula['synopsis'];
    $p->save();
}
```

Exercise

- Seeds

Since you are going to use a model, remember that you will have to import that model at the beginning of the DatabaseSeeder file.

Run the Artisan command to process the seeds and then check in phpMyAdmin that the data has been entered correctly.

Exercise

- Database usage

At this point the system is already prepared to be able to use the database. Thus, the CatalogController controller code must be modified:

✓ Modify the getIndex method to get the list of movies using the Movie model and pass it to the view.

✓ Modify the getShow method to get the movie based on its id. You must use the findOrFail method and pass the data to the view

✓ Modify the getEdit method to get the movie based on its id. You must use the findOrFail method and pass the data to the view.

✓ Remove the movie array that was introduced in the controller

Exercise

- Database usage

The next step is to update the views so that the data is obtained from the object with the movie they receive now. We will have to change the syntax from $movie['field'] to $movie->field.

In the index.blade.php view, instead of using the array index to create the link, you will have to use the id of the movie: $pelicula->id. The same will have to be done in the show.blade.php view to generate the edit movie link