

# Laravel: Eloquent ORM



Juan Carlos Pérez Rodríguez

## Sumario

Uso y configuración básica de sqlite en Laravel.....	4
Creación de Modelo y Migraciones en Laravel.....	5
Primeros pasos con Eloquent.....	11
Generar las clases del modelo tomando la información de las tablas.....	11
Detalles a revisar en las entidades generadas.....	13
Eloquent find().....	15
Obtener todos los objetos de una tabla: método all().....	16
Collect.....	17
.....	18
Guardar y borrar en una única tabla.....	19
Caso guardar y borrar en una única tabla que no tiene foreign keys.....	19
Mediante objetos.....	19
Con array clave/valor de campos a guardar en la tabla.....	21
Gestionando relaciones entre tablas.....	24
Caso Entity en relación N:1.....	24
Grabando con N:1.....	25
associate().....	26
Uso de refresh().....	27
Caso Entity en relación 1:N.....	28
Grabando con 1:N.....	28
Caso relaciones N:M.....	29
Caso queremos una Entity que refleje la tabla intermedia.....	29
Caso NO queremos una Entity para la tabla intermedia.....	30
Grabando con N:M.....	30
Gestión de transacciones.....	32
Manejo de fechas con eloquent laravel.....	34
Consultas avanzadas.....	35
Autenticación.....	37
Creando la tabla users con migraciones.....	38
Usando el Middleware de Autenticación.....	39
Creando Middleware.....	42
Anexo: Sustituir la clase User por una personalizada.....	44
Modificando el registro de un usuario.....	47
Modificando el login de un usuario.....	48

Juan Carlos Pérez Rodríguez

## Uso y configuración básica de sqlite en Laravel

Vamos a hacer uso de sqlite ( php tiene todo su manejo ya preinstalado ) para hacer nuestras primeras pruebas con Eloquent

SQLite es un SGBD relacional que ( lo siguiente es tomado de wikipedia ):

“no es un proceso independiente con el que el programa principal se comunica. En lugar de eso, la biblioteca SQLite se enlaza con el programa pasando a ser parte integral del mismo”

Así pues, queda clara su ligereza y para aplicaciones que lo que quieren es una base de datos casi integrada en la aplicación sin muchos requerimientos es bastante apropiada

Crearemos una nueva base de datos sqlite para nuestra aplicación en laravel simplemente ejecutando el comando touch sobre la ruta database de nuestra aplicación:

(supondremos que estamos ubicados en la carpeta raíz de nuestra aplicación laravel )

```
touch database/database.sqlite
```

Ahora vamos a especificar en laravel que haga uso de sqlite sobre el fichero recién creado

Lo primero es agregar/modificar el fichero oculto: `carpetadelproyectolaravel/.env`

con los datos pertinentes ( comentaremos todas las líneas que comiencen con DB\_ para luego volver a reestablecer ya que por defecto está configurado para mysql )

las líneas que pondremos:

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
```

(donde /absolute/path/to hace referencia al path completo hacia el fichero que creamos con touch )

**Nota:** Si no se pone la ruta absoluta puede fallar cuando luego lancemos la aplicación web con: php artisan serve

Ahora modificaremos el fichero:

`config/database.php`

para que soporte en sqlite foreign keys:

```
'sqlite' => [  
    // ...  
    'foreign_key_constraints' => true,  
],
```

Algo que puede sernos bastante útil es una aplicación que nos ayude a trabajar con sqlite. Una alternativa rápida es el uso de una extensión en el navegador. Por ejemplo el addon de visualstudio: **Sqlite**

## Creación de Modelo y Migraciones en Laravel

Bien, ya estamos en el punto de crear nuestros ficheros-modelo para luego hacer uso de eloquent y generar las tablas pertinentes en nuestra base de datos. Para ello haremos uso del comando artisan:

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

Observar el parámetro: “-m” mediante esa opción le estamos diciendo que prepare las migraciones para ese objeto del modelo ( que prepare para luego llevar la clase a una tabla de la base de datos ) ESTO ÚNICAMENTE PREPARA LA MIGRACIÓN NO LA EJECUTA

● **Práctica 1:** Crear un elemento del modelo mediante php artisan para luego guardarlo en sqlite. En concreto crearemos: Alumno ( para esta prueba lo único que guardaremos será el nombre apellidos y edad )

Debe habernos creado algo parecido a:

```
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Alumno extends Model  
{  
    //  
}
```

Para crear la tabla en la base de datos usamos **php artisan migrate** ( lo usaremos en breve )

Presumiblemente se creará en la base de datos un campo **id** y los **timestamps**, cosa que podemos comprobar abriendo la base de datos sqlite desde la aplicación que viene como un addon en el navegador u otra aplicación para sqlite.

Nosotros precisamos tener información del nombre del producto y el precio así que habrá que establecerlo como atributos:

```
class Alumno extends Model
{
    /**
     * @var string $nombre
     */
    protected $nombre;
    /**
     * @var int $edad
     */
    protected $edad;
}
```

una vez ejecutado podemos editar el archivo de migración creado en la carpeta:

**carpetadelproyectolaravel/database/migrations**

Podemos editar ese fichero. Veamos un ejemplo:

```

return new class extends Migration
{
/**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
Schema::create('alumnos', function (Blueprint $table) {
    $table->id();
    $table->timestamps();
    $table->string('nombre',100)->nullable(false);

});
}

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
Schema::dropIfExists('alumnos');
}
};

```

Se ha marcado como hemos agregado un campo string de tamaño 100 y que no puede ser nulo. Podríamos haber dicho que fuera de tipo: **integer** o haber establecido un valor por defecto

Ejemplo:

```
$table->string('talla',10)->default('X')
```

Ahora para ejecutar la migración y afecte la database:

```
php artisan migrate
```

● **Práctica 2:** Realizar la migración para Alumno ( para esta prueba lo único que guardaremos será el nombre apellidos y edad ) Comprobar mediante el add-on sqlite que se ha creado en la database

Para establecer campos adicionales después de un primer fichero de migración ( puede haber varios sucesivos ) que queramos en la base de datos debemos crear un fichero de migración para establecerlos y luego ejecutar la migración ( se ejecutan por orden las migraciones de los diferentes ficheros que hagamos).

Para eso vamos a usar el comando artisan:

```
php artisan make:migration add_columns_to_productos_table --table=productos
```

El comando anterior le dice que agregue un nuevo fichero de migración ( que no afecta a los anteriores y así se ejecuta sin problemas la creación de la tabla productos que teníamos creada previamente ) con el nombre: add\_columns\_to\_productos\_table ese nombre nos resulta significativo de que queremos agregar nuevos campos a nuestra tabla. Finalmente le decimos con el comando anterior: “--table=productos” de esa forma le estamos diciendo que el fichero de migración que nos fabrique sea para la tabla productos

Lo más probable es que nos haya creado un código similar al siguiente:

```
class AddColumnsToProductosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('productos', function (Blueprint $table) {

        });
    }
}
```



```

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::table('productos', function (Blueprint $table) {

        });
    }
}

```

En la function: up(), en la function dentro de Schema::table() establecemos los campos que queremos crear. Por ejemplo, si queremos crear un campo varchar que se llame: “nombre” y sea de tamaño 100:

```

class AddColumnsToProductosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('productos', function (Blueprint $table) {
            $table->string('nombre',100)->nullable(false);
            $table->string('categoria',80)->default('X');
        });
    }
}

```

Observar que hemos puesto también: `nullable(false)` eso significa que el campo no puede ser nulo. También se especifica un valor por defecto con: `default()`

En el caso que queramos hacer un rollback de nuestros cambios en la base de datos y dejarlo todo como estaba debemos deshacer los cambios en la función: down() Quedando tal función así:

```

public function down()
{
    Schema::table('productos', function (Blueprint $table) {
        $table->dropColumn(['nombre','categoria']);
    });
}

```

```
}
```

Se ve que le pedimos que elimine la columna llamada: “nombre” y ‘categoria’.  
**IMPORTANTE** no se escriben varias líneas `dropColumn()` Una única sola

Para que la migración anterior tenga efecto en la base de datos ( recordar que ya se ha hecho previamente una migración y ahora toca actualizar ) ejecutaremos:

```
php artisan migrate:refresh
```

El comando anterior le dice que recree las tablas en la base de datos con la información de migraciones que tenemos ahora. La ejecución de los comandos descritos harán que exista el campo nombre dentro de la tabla productos

Si hubiéramos realizado el fichero de migración aparte de cuando creamos el fichero del modelo ( recordar que pusimos la opción: “-m” en el `make:model` ) La sentencia para crear la tabla productos mediante una migración habría sido:

```
php artisan make:migration migracionproductos --create productos
```

Observar que la opción: `--create` nos genera en el fichero de migración: `Schema::create`

Hay diversos comandos para crear los diferentes tipos de datos en la base de datos( integer, enum, etc ) para revisarlos podemos ir a la documentación oficial:

<https://laravel.com/docs/5.6/migrations#creating-columns>

● **Práctica 3:** Haciendo uso de la documentación oficial y ejecutando las migraciones Crear una tabla Productos con string nombre y también un campo llamado precio que soporte precios con decimales. Así como un campo cantidad que represente la cantidad de producto de la que disponemos

● **Práctica 4:** Esta es una práctica de autoformación. Buscar como hacer uso de los: “seeder” y rellenar datos aleatorios en la tabla productos de la base de datos con ese sistema

## Primeros pasos con Eloquent

### Generar las clases del modelo tomando la información de las tablas

Lo primero es que vamos a volver a Mysql como nuestra SGBD. Y establecemos la base de datos que contenga la información para nuestra tienda online. Esto es, editaremos el fichero oculto .env: `baseproyectolaravel/.env` y especificamos la parte de bases de datos ( variables con DB\_ como prefijo )

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=instituto
DB_USERNAME=root
DB_PASSWORD=1q2w3e4r
```

Ahora ejecutamos en consola ( en la base del proyecto):

```
composer require krlove/eloquent-model-generator --dev
```

Hay que registrarlo como un provider. Para ello nos vamos a: `config->app.php` buscamos la parte de providers y agregamos:

```
Krlove\EloquentModelGenerator\Provider\GeneratorServiceProvider::class
```

En la siguiente imagen aparece el array de providers donde hemos agregado la línea anterior:

( recordar fichero config/app.php )

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Notifications\NotificationServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords>PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,

    /*
     * Package Service Providers...
     */

    /*
     * Application Service Providers...
     */
    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    // App\Providers\BroadcastServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,
    Krlove\EloquentModelGenerator\Provider\GeneratorServiceProvider::class,
```

Y después por cada tabla que queramos traernos ( en el ejemplo nos traemos una tabla Producto )

```
php artisan krlove:generate:model Producto --table-name=productos
```

**nota:** Si no queremos los timestamps ( es posible no los hayamos puesto en nuestra base de datos):

```
php artisan krlove:generate:model Producto --table-name=productos --no-timestamps
```

● **Práctica 5:** Hacer los cambios pertinentes en: .env para que la aplicación use la base de datos mysql con las tablas pertinentes para instituto matrículas. Crear con generate:model las clases correspondientes

## Detalles a revisar en las entidades generadas

Si vemos la documentación de krlove nos nombra que podemos generar todas las entities con una única instrucción:

```
php artisan krlove:generate:models
```

Pero ahí no tenemos la posibilidad de especificar el nombre de la tabla de la base de datos. Así pueden darse errores respecto a los nombres que piensa que deba encontrarse y producirse fallos. Es preferible hacerlo de forma controlada una a una, especificando el nombre de la tabla y si lleva o no timestamps

Precisamente por esa disparidad de las cosas esperadas, tenemos que revisar nuestros modelos generados. Por ejemplo, laravel se espera encontrar un campo id que será la primary key. Veamos un ejemplo con Alumno:

```
class Alumno extends Model
{
    protected $table = alumnos;
    public $timestamps = false;

    /**
     * The primary key for the model.
     *
     * @var string
     */
    protected $primaryKey = 'dni';

    /**
     * The "type" of the auto-incrementing ID.
     *
     * @var string
     */
    protected $keyType = 'string';

    /**
     * Indicates if the IDs are auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
```

Vemos que se tiene que especificar el nombre de la primaryKey porque no es el que se espera. En este caso se llama dni: `$primaryKey = 'dni'`

Como presupone un id autoincremental le tenemos que decir que realmente no es autoincremental y que tampoco es numérico sino una string: `$keyType = 'string'`

```
$incrementing = false;
```

Comentemos ahora: `$table=alumnos` Observar que para el caso entity: Alumno y tabla: alumnos Laravel lo va a hacer bien y se va a dar cuenta que tabla corresponde a la entity. Pero si nosotros lo especificamos en el entity mediante: `$table` nos garantizaremos siempre que hará correctamente el mapeo entre la entity y la tabla

Finalmente debemos tener en cuenta que con Eloquent, la primary key NO puede ser compuesta. En el caso, por ejemplo, de una tabla intermedia en una relación N:M habrá de ponerse un campo único que hará de primary key ( típicamente un autoincremental ). Aunque siempre podremos especificar que los campos que de otra forma habrían sido una clave compuesta, tengan la restricción unique e indexarlos

## Eloquent find()

¿ cómo tomamos de la base de datos el producto con el id dado? Las colecciones en Laravel contemplan las búsquedas por la primarykey mediante la función: `find()` Así podríamos buscar:


```
$miproducto = Producto::find(23);
```

Y nos devolvería el producto con primarykey 23

**Nota:** si estamos volcando directamente un objeto sin parsear sus atributos. Posiblemente sea buena idea hacer un `json_encode`. Ejemplo para convertir una moneda a una string json:

```
json_encode($moneda, JSON_UNESCAPED_UNICODE)
```

Observar que hacemos que nos muestre los símbolos castellanos de una forma limpia con la bandera: `JSON_UNESCAPED_UNICODE`

-  **Práctica 6:** Construir la ruta para las peticiones GET a: `veralumno` que nos lleve a un controlador Este controlador hará la búsqueda por id del alumno y lo enviará a una vista que mostrará la información

## Obtener todos los objetos de una tabla: método all()

Eloquent tiene varias formas para recuperar objetos de la base de datos. De momento vamos a ver una en la que se toma toda la colección de objetos para una tabla de la base de datos:

```
function listar(){  
  
    $productos = Productos::all();  
  
    foreach ($productos as $producto) {  
        echo "<br>". $producto;  
        //var_dump($producto->attributesToArray());  
    }  
}
```

Vemos la función: `all()` mediante esa función obtenemos una colección de todos los productos almacenados en la tabla productos. También vemos que para la instancia del modelo `$producto` podemos usar: `attributesToArray()` y que nos devuelva un array con todos los datos de la fila de la tabla

● **Práctica 7:** Listar todos los alumnos de nuestra base de datos ( debe mostrarse en una vista blade )



## Collect

Eloquent ( el ORM de laravel ) siempre devuelve sus conjuntos de resultados como Collection que son un objeto de laravel cómodo para trabajar:  
<https://laravel.com/docs/5.8/collections>

dispone entre otros de:

- **count()** que nos dice la cantidad de objetos en la colección:

```
$productos = Producto::all();  
$totalDeProductos = $productos->count();
```

- **contains()** que nos sirve para buscar values :

```
$collection = collect(['name' => 'Desk', 'price' => 100]);  
$verdaderoFalso = $collection->contains('Desk'); //devuelve un boolean
```

o buscar un par clave-value:

```
$collection->contains('product', 'Bookcase');
```

Podemos hacer filtrados con **where()**, ordenamientos con **orderBy()**, limitar la cantidad de resultados con **take()**. Ejemplo: Obtener los 2 productos más caros con coste menor a 300:

```
$productos = Producto::where('precio','<',300)  
->orderBy('precio','desc')  
->take(2)  
->get();
```

La orden: **get()** nos devuelve **el conjunto de resultados** solicitado. También existe la orden **first()** que nos devuelve **el primer objeto** que cumple los requisitos solicitados. Así si escribiéramos:

```
$productos = Producto::where('precio','<',300)  
->orderBy('precio','desc')  
->take(2)  
->first();
```

Obtendríamos el producto con el mayor precio, siempre que cueste menos de 300.

● **Práctica 8:** Buscar mediante where y mostrar en una vista las matrículas anteriores a 2021

● **Práctica 9:** continuando con la anterior, ordenaremos las matrículas por fecha y ejecutaremos un `take(1)` ( seguir el ejemplo de encima: `:where(..l)->orderBy(...)->take(1)` ) y finalmente terminaremos con un `get()` Mediante `var_dump()` veremos que estructura nos devuelve. Hacer lo mismo de nuevo pero ahora en lugar de terminar con un `get()` terminamos con un `first()` ¿ da estructuras diferentes ? Nota: observar que la idea de `first` es tomar un objeto y `get` era el conjunto de objetos

Ya hemos nombrado: **`count()`** que deja de devolvernos Objetos del modelo y nos devuelve un escalar que denota la cantidad de objetos en el colección. También disponemos de: **`sum('nombrecolumna')`** y **`max('nombrecolumna')`** que nos devuelven la suma de los datos de la columna especificada por: `nombrecolumna` y el máximo de esos datos respectivamente.

● **Práctica 10:** Obtener la cantidad total de asignaturas de 1ºDAM y mostrarlo

Tiene mucho más potencial `collect`. Se pueden hacer `join`, soporta `lambda` con `foreach` , filtrado, `flatMap`, etc

## Guardar y borrar en una única tabla

**Nota:** Si no tenemos timestamps en la DDBB es importante que especifiquemos en nuestras clases modelo que no los tenemos porque de otra forma tendremos problemas en el guardado

```
class Alumno extends Model
{
    public $timestamps = false;
```

## Caso guardar y borrar en una única tabla que no tiene foreign keys

### Mediante objetos

Vamos a generar una moneda mediante el lenguaje de objetos, con el método: `save()` y luego borraremos con `delete()`

Probar el siguiente código en un método del controlador:

```
$mon = new Moneda();
$mon->nombre = "dolar2";
$mon->pais = "Zirpe";
$mon->save();

$monedas = Moneda::all();
foreach ($monedas as $monedakey => $moneda) {
    echo json_encode($moneda, JSON_UNESCAPED_UNICODE) . "<br>";
}

$mon1 = Moneda::where('nombre','="','dolar2')->first();
$mon1->delete();

die;
```

Veamos otro ejemplo de como generar un objeto: Para un controlador de productos guardaremos el objeto en la database:

```
class GestionarProductos extends Controller{

    function guardar(){
        /**
         * @var Productos $producto
         */
        $producto = new Productos();
        $producto->nombre = "unproducto";
        $producto->precio = 111;

        echo "Ejecutando el grabado del producto";
        var_dump($producto);
        $producto->save();
        /**
         * Productos::create([
         *     'nombre' => 'mipapit',
         *     'precio' => 80,
         * ]);
         */
    }
}
```

Vemos que hemos usado la anotación @var para la variable local \$producto. Así el ide nos ayuda con el autocompletado. Por otro lado vemos la línea importante de cada a la persistencia:

```
$producto->save();
```

Con el comando anterior hemos guardado en la base de datos el objeto Producto. La orden `save()` nos permite tanto creación como actualización de un objeto en la base de datos ( eloquent se da cuenta que el objeto ya está creado y simplemente lo modifica al escribir: `$producto->save()` )

## Con array clave/valor de campos a guardar en la tabla

Para este caso se hace uso del método `create()` que nos proporciona laravel:

Entre comentarios en el ejemplo de antes vemos otro trozo de código:

```
Productos::create([
    'nombre' => 'mipapit',
    'precio' => 80,
]);
```

Se está llamando al método estático `create()` de la clase `Productos` estableciendo como parámetro un array clave/valor. Esa es otra forma perfectamente válida para grabar en la base de datos en lugar de: `$producto->save()` En cualquier caso, para hacer uso de: `create()` debemos establecer en el modelo que esos datos son: “fillable”, ya que Eloquent hace uso de protección en caso de que un usuario 'malicioso' haya modificado los campos del formulario. Para ello nosotros tenemos que especificar por ejemplo:

```
class Productos extends Model
{
    protected $table = 'productos';

    protected $fillable = ['nombre', 'precio'];
}
```

Fijémonos en: `protected $table = 'productos';` Gracias a ese trozo de código garantizamos que Eloquent nos relacione nuestra clase `Productos` con la tabla `productos` de la base de datos ( por defecto Eloquent busca las similitudes de los nombres pero con esta opción pueden ponerse nombres completamente diferentes )

Y volviendo con `fillable`, también observamos que, el usuario puede pasarle a eloquent ( y que quede grabado en la base de datos ) la información de nombre y precio, pero no se guardará la información de los timestamps ( `created_at`, `updated_at` que se crearon también al hacer las migraciones )

La anterior política definida para “fillable” es al estilo de una white-list. Esto es, se especifican los parámetros que son válidos para rellenar y guardar en la base de datos dados por el usuario. Sin embargo existe la política de una black-list. En ese caso todo parámetro se puede guardar en la base de datos salvo los que se especifiquen mediante la palabra: `guarded` Así, para usar esa política tendríamos que especificar en la clase `Producto`:

```
class Productos extends Model
{
    protected $table = 'productos';

    protected $guarded= [];
```

Con lo anterior estamos diciendo que se acepta guardar la información para todos los campos excepto los que se han establecido en el array: \$guarded pero como ese array está vacío significa que se aceptan TODOS

En el caso de moneda tenemos un fillable con nombre y pais. En ese caso veamos como habría sido con array clave/vañpr:

```
$mon = new Moneda();
$mon->nombre = "dolar2";
$mon->pais = "Zirpe";
// $mon->save();

Moneda::create([
    'nombre' => $mon->nombre,
    'pais' => $mon->pais
]);

$monedas = Moneda::all();
foreach ($monedas as $monedakey => $moneda) {
    echo json_encode($moneda, JSON_UNESCAPED_UNICODE) . "<br>";
}

$mon1 = Moneda::where('nombre','="', 'dolar2')->first();
$mon1->delete();

die;
```

Más información oficial: <https://laravel.com/docs/5.8/eloquent#mass-assignment>

● **Práctica 11:** Crear 2 nuevas asignaturas Siguiendo lo que se detallado: Una de 1ºDAM mediante new Aasignatura() y save() y otra de 2ºDAM mediante: Aasignatura::create() Se debe haber especificado mediante: fillable los campos que se aceptan para rellenar en la DB.

● **Práctica 12:** Modificar las dos asignaturas anteriores de tal forma que la que era de primero pase a segundo y viceversa. Finalmente borrar de la base de datos la que sea de segundo

Juan Carlos Pérez Rodríguez

# Gestionando relaciones entre tablas

## Caso Entity en relación N:1

Si nos fijamos en las clases del modelo que nos ha creado laravel a partir de las tablas de la base de datos y nos fijamos en la clase Matrícula, podemos ver como trata las relaciones con otras tablas-clases. Por ejemplo, sabemos que matriculas tiene una foreign-key del alumno:

```
class Matricula extends Model
{
    public $timestamps = false;
    /**
     * The primary key for the model.
     *
     * @var string
     */
    protected $primaryKey = 'idmatricula';

    /**
     * @var array
     */
    protected $fillable = ['dni', 'year'];

    /**
     * @return \Illuminate\Database\Eloquent\Relations\BelongsTo
     */
    public function alumno()
    {
        return $this->belongsTo('App\Models\Alumno', 'dni', 'dni');
    }
}
```

Observamos que nos ha creado una función: usuario() que hace uso de: belongsTo() Lo bueno de esta forma de trabajar de Eloquent es que cuando nosotros ponemos:

```
$mat = Matricula::first();
```

```
$alumno = $mat->alumno;
```

Se obtiene el alumno como si estuviéramos accediendo a una propiedad ( cuando nosotros lo que hemos creado es un método llamado: alumno() )



Si observamos la función belongsTo() entendemos como está haciendo las relaciones eloquent:

```
belongsTo('App\Usuario', 'fk_usuario', 'idusuario')
```

La anterior línea dice que es un objeto de una clase modelo: Usuario y donde nosotros ponemos en otra tabla-modelo el campo: “fk\_usuario” está vinculado y debe ser igual al campo “idusuario” de la tabla-modelo Usuario. Siempre ponemos primero la foreign key y luego el campo origen

## Grabando con N:1

En el siguiente ejemplo, se presupone que tenemos una moneda con id=1 que es un dólar. Y vamos a guardar un nuevo tipo de cambio para la fecha 2021-12-31 a 0.89€. Observar que estamos poniendo el número de id de la foreign key directamente, NO el objeto moneda

```
$dolar1 = Moneda::find(1);
echo json_encode($dolar1, JSON_UNESCAPED_UNICODE) . "<br>";
foreach( $dolar1->historicos as $h){
    echo json_encode($h, JSON_UNESCAPED_UNICODE) . "<br>";
}
echo "<br>";

$historicoNuevo = Historico::create([
    'fkidmoneda'=>1,
    'fecha'=>'2021-12-31',
    'equivalenteeuro'=>0.89
]);
//$dolar1->refresh();

echo "creado: ".json_encode($historicoNuevo, JSON_UNESCAPED_UNICODE) . "<br>";

echo json_encode($dolar1, JSON_UNESCAPED_UNICODE) . "<br>";
foreach( $dolar1->historicos as $h){
    echo json_encode($h, JSON_UNESCAPED_UNICODE) . "<br>";
}
```

Al ejecutar el código anterior comprobamos que la información de históricos alcanzada desde moneda: \$moneda->historicos no se actualiza en RAM después de haber creado el histórico para el día 2021-12-31.

● **Práctica 13:** Basándose en el ejemplo anterior ( usando Historico::create ) crear un histórico para la moneda dólar que sea para la fecha actual al tipo de cambio actual con el euro

Lo anterior lo podríamos haber ejecutado también con una orden save() de una Entity Historico nueva:

```
$historicoNuevo = new Historico();  
$historicoNuevo->fkidmoneda = 1;  
$historicoNuevo->fecha = '2021-12-31';  
$historicoNuevo->equivalenteeuro = 0.89;  
$historicoNuevo->save();
```

Importante: observar que de nuevo **NO** asignamos un objeto Moneda en el campo de foreign key, de nuevo tenemos que poner el **id** correspondiente en su lugar

Para poder usar plenamente el lenguaje de objetos en el caso anterior, debemos usar:

### *associate()*

Mediante associate() podemos establecer la foreign key que corresponda en nuestra tabla-entity

Ejemplo:

```
$dolar1 = Moneda::find(1);  
$historicoNuevo = new Historico();  
$historicoNuevo->fecha = '2021-12-31';  
$historicoNuevo->equivalenteeuro = 0.89;  
$historicoNuevo->moneda()->associate($dolar1);  
$historicoNuevo->save();
```

Observar que NO usamos un atributo moneda: \$historicoNuevo → moneda sino un método moneda: \$historicoNuevo → moneda()

● **Práctica 14:** crear un histórico para la moneda dólar que sea para la fecha de mañana con tipo de cambio con el euro actual menos un céntimo, usando save() y associate()

Tenemos una forma adicional, que consiste en crear ( ejecutar save() ) desde la Entity realacionada ( la que está en la relación como 1:N )

Veamos ejemplo:

```
$historicoNuevo = new Historico();
$historicoNuevo->fecha = '2021-12-31';
$historicoNuevo->equivalenteeuro = 0.89;
$dolar1->historicos()->save($historicoNuevo);

//$dolar1->refresh();
echo "generado: ".json_encode($historicoNuevo, JSON_UNESCAPED_UNICODE) . "<br>";
```

En el ejemplo vemos que hemos introducido en el nuevo objeto Historico los otros campos que no son la foreign key. Luego ejecutamos desde la Entity Moneda el save():

```
$dolar1->historicos()->save($historicoNuevo);
```

Observar que **NO** accedemos a los históricos como un atributo: \$dolar1->historicos sino como un método: \$dolar1->historicos() en otro caso nos dará error

● **Práctica 15:** crear un histórico para la moneda dólar que sea para la fecha de pasado mañana con tipo de cambio con el euro actual menos dos céntimos usando save() desde la entidad Moneda ( siguiendo el ejemplo que acabamos de ver )

## Uso de refresh()

Tenemos comentada una línea con un refresh() . Si usamos refresh() forzamos a que la entity que tenemos en ram se actualice con la información almacenada en la base de datos

Probar a quitar el comentario anterior y volver a ejecutar. Observaremos que al hacer:

\$dolar1 → refresh() se carga la información del nuevo histórico generado en la lista de históricos de la moneda \$dolar1. Observar que eso significa que adicionalmente a haber buscado en la tabla monedas para actualizar la información de la Entity Moneda, también ha buscado en sus tablas relacionadas y ha encontrado que hay un nuevo histórico

## Caso Entity en relación 1:N

Observamos que belongsTo() informa de los campos muchos a 1 → N:1

Para cuando queremos usar el uno a muchos → 1:N hacemos uso de hasMany()  
Observemos el trozo de código que ha generado laravel para los detalleCarrito en una clase Carrito ( pensar en una tienda con un carrito y los detallescarrito ):

```
/**
 * @return \Illuminate\Database\Eloquent\Relations\HasMany
 */
public function detalleCarritos()
{
    return $this->hasMany('App\DetalleCarrito', 'fkidcarrito', 'idcarrito');
}
```

se ha creado una función: detalleCarritos() pero si queremos podemos tomarlo como una propiedad de Carrito:

```
$carrito = Carrito::first();
$coleccionDetalleCarrito = $carrito->detalleCarritos;
```

La función hasMany() se interpreta así:

```
hasMany('App\DetalleCarrito', 'fkidcarrito', 'idcarrito')
```

Cada carrito tiene muchos DetalleCarrito. El siguiente parámetro siempre es la foreign key así que fkidcarrito hace referencia en este caso al nombre que aparece en la tabla-clase DetalleCarrito. El tercer campo es el origen de la relación, en este caso sería nuestro idcarrito

Podemos observar que tanto en hasMany() como en belongsTo() se pone primero la foreign key y luego el campo origen

## Grabando con 1:N

No hay nada especial que agregar para este caso. Como no hay campos foráneos, simplemente rellenamos los campos de nuestra Entity y ejecutamos: **save()**.

## Caso relaciones N:M

Lo primero de todo, tener en cuenta que **Eloquent no admite claves primarias compuestas**. Así que, una Entity que refleje la tabla intermedia de una relación N:M no puede tener como clave primaria la composición de los ids de las otras dos tablas. Una buena alternativa puede ser una clave primaria que sea autoincremental y hacer que tengan una restricción unique las dos claves foráneas

## Caso queremos una Entity que refleje la tabla intermedia

Podemos tratar ese caso simplemente como una Entity con dos relaciones N:1 . Esto es: tendrá dos belongsTo() uno para cada una de las entidades relacionadas. Por supuesto en las otras dos entidades estará el correspondiente hasMany()

Así cuando queramos generar un objeto de esa entidad intermedia ejecutaremos el correspondiente: `new NombreEntity()` y antes del: `save()` estableceremos los dos `associate()` ( o si se prefiere no usar el lenguaje de objetos, un: `NombreEntity::create()` con sus correspondientes dos ids para las claves foráneas )

## Caso NO queremos una Entity para la tabla intermedia

Para ese caso debemos usar: `belongsToMany()` Veamos como ejemplo, la relación entre Matrícula y Asignatura. Sabemos que es una relación N:M ya que en una matrícula hay múltiples asignaturas y cada asignatura puede estar en múltiples matrículas. En ese caso en la Entity Matricula podemos crear un método asignaturas así:

```
public function asignaturas()
{
    return $this->belongsToMany(
        Asignatura::class, //objetos de la relación manytomany que queremos obtener
        'asignatura_matricula', //nombre de la tabla de enlace
        'idmatricula', // aquí el nombre de la foreign key en la tabla de enlace de nuestra entity id
        'idasignatura' // aquí la foreign key de tabla de enlace, que apunta a la otra entidad
    );
}
```

### Grabando con N:M

Imaginemos que estamos creando una nueva matrícula, primero ejecutamos un `save()` para que persista en la tabla matriculas y luego un `save()` por cada asignatura. El siguiente ejemplo crea una nueva matrícula para el primer alumno que se encuentre, para el año 2023. Luego genera en la tabla intermedia la información que enlaza la matrícula con las asignaturas de esa matrícula

```
$alumno = Alumno::all()->first();
$nuevaMatricula = new Matricula();
$nuevaMatricula->alumno()->associate($alumno);
$nuevaMatricula->year = 2023;
$nuevaMatricula->save();
foreach( Asignatura::all()->take(2) as $asignatura){
    $nuevaMatricula->asignaturas()->save($asignatura);
}

$nuevaMatricula->refresh();
echo json_encode($nuevaMatricula, JSON_UNESCAPED_UNICODE) . "<br>";
```

Lo especial aquí es que llamamos al método que construimos con el belongsToMany() para ejecutar el save(): `$nuevaMatricula → asignaturas() → save($asignatura)` de esa forma eloquent sabe que debe guardar la relación entre la matrícula y la asignatura

● **Práctica 16:** crear entre asignatura y matrícula una relación N:M sin generar entity para la tabla intermedia. Tanto desde matrícula como desde asignatura. Luego mostrar por cada asignatura que personas ( su nombre y su dni ) se han matriculado

Para el caso que mostramos antes de, guardar una matrícula y luego los insert en la tabla intermedia, debemos poder restituir las cosas como estaban inicialmente si no se puede registrar alguna asignatura para la matrícula dada ( deshacer los insert realizados ) para eso podemos usar las transacciones.

## Gestión de transacciones

Gestionamos las transacciones mediante: `DB::beginTransaction()` , `DB::commit()` y `DB::rollBack()`;

```
DB::beginTransaction();
try{
    $model1 = new Type();
    $model1->test = 'great';
    $model1->save();

    $model2 = new Type();
    $model2->test2 = 'awesome';
    $model2->save();
    DB::commit();
}catch(Exception $e){
    DB::rollBack();
}
```

Si no queremos poner nosotros los comando: `DB::commit()` y `DB::rollBack()` tenemos la opción de ponerlo dentro de una función llamada transaction: `DB::transaction()`

En el siguiente ejemplo se guarda todo junto: la matrícula y las asignaturas para esa matrícula. Vemos también que se dispara un rollback si la asignatura que se ha insertado es “BAE” en cuyo caso se deshace toda la transacción

```
DB::transaction(function () {
    $alumno = Alumno::all()->first();
    $nuevaMatricula = new Matricula();
    $nuevaMatricula->alumno()->associate($alumno);
    $nuevaMatricula->year = 2023;
    $nuevaMatricula->save();

    $asignatura = Asignatura::first();

    $nuevaMatricula->asignaturas()->save($asignatura);
    if( $asignatura->nombre == "BAE")
        DB::rollBack();
});
```



● **Práctica 17:** crear un miniformulario donde se introduzca una moneda y un histórico. Utilizar una transacción de tal forma que si al guardar un histórico que no tiene un tipo de cambio numérico sino que el usuario ha introducido texto, se deshaga el guardado de los dos objetos

Juan Carlos Pérez Rodríguez

## Manejo de fechas con eloquent laravel

```
//declaramos este atributo para guardar la fecha en la tabla DDBB
protected $dates = [ 'fecha' ];

public function setFechaAttribute( $value){
//esta función toma una string para guardar en la base de datos
$this->attributes['fecha'] = (new Carbon($value))->format('Y-m-d H:m:s');
}

//esta función toma la info de la DDBB y lo muestra
public function getFechaAttribute( $value){
    return Carbon::parse($value)->format('d/m/Y H:m');
}
```

Para el correcto funcionamiento debemos poner en el atributo array: \$dates los campos que son fechas ( en el ejemplo es únicamente 'fecha' ) Y se crean los getter y setter para cada uno de esos campos: setFechaAttribute(), getFechaAttribute() en nuestro caso. Mediante:

\$this->attributes[] podemos alcanzar las propiedades de tipo fecha de nuestro modelo, en nuestro caso lo hacemos para: \$this->attributes['fecha'] A ese atributo le debemos asignar un objeto de tipo Carbon que es la forma en la que Laravel maneja las fechas. La sentencia use para Carbon es:

```
use Carbon\Carbon;
```

## Consultas avanzadas

**Podemos filtrar con where en la propia tabla-objeto y con wherehas en las tablas relacionadas con foreign key**

Imaginemos el caso de carrito compra ( equivalente a un pedido ) y los detalles de ese carrito ( equivalente a detalle de pedido ) Es lógico que el pedido/carrito tenga un propietario ( un usuario ) .

En la anterior situación puede ocurrir que queramos saber cuantas veces un usuario ha guardado en su carrito ( ha comprado si hace el pedido ) determinado producto. De esa forma sabremos que productos le son especialmente atractivos

Para esa consulta debemos tener en cuenta que un usuario tiene carritos y en cada carrito hay detalle\_carrito y podemos filtrar los detalle\_carrito para un usuario con id 7 y un producto con id 23 mediante:

```
$idproducto = 23;
$idusuario = 7;

$detalles = Detalle_carrito::wherehas(
    'carrito.usuario',
    function($query) use($idusuario){
        $query->where('idusuario',$idusuario);
    }
)
->where('idproducto',$idproducto)
->get();
```

Observar que le estamos diciendo que queremos filtrar los detalles de carrito que su tabla relacionada: Carrito tenga un campo usuario donde coincida con el idusuario que nosotros le pasamos. **wherehas()** permite poner sentencias where y limitar el conjunto de resultados basándose en que exista la condición where en la tabla relacionada ( en otras palabras, hacer consultas anidadas ) Para poder informar del campo que queremos hacer la consulta anidada hacemos uso del punto: “.” así si ponemos: “carrito.usuario” informamos que queremos hacer un filtro a el objeto usuario para cada carrito, **Importante:** en la clase: **Detalle\_carrito** debe haber un atributo: **carrito**, luego nos vamos a la clase: **Carrito** allí buscamos y vemos que hay un atributo: **usuario** Es por eso que en el wherehas escribimos: ‘**carrito.usuario**’

Otra alternativa es usar consultas sql nativas, mediante: [DB::select\(\)](#) (ya no es Eloquent)  
Este ejemplo es de la documentación oficial:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Observar que se hace una sentencia sql estandar con parámetros definidos por los dos puntos: “:” y luego se le pasa un array clave/valor para identificar esos parámetros

Laravel en su documentación avisa del peligro de sql-injection si usamos sql nativo

## Autenticación

A partir de la versión 6 el módulo de autenticación hay que instalarlo aparte. Vamos a la dirección de la carpeta que creamos con el comando anterior:

```
cd nombreproyecto;  
composer require laravel/breeze  
php artisan breeze:install  
npm install  
composer require doctrine/dbal
```

Con los comandos anteriores, por ejemplo, nos genera en las vistas: auth, home.blade.php, layouts . La última línea sirve para hacer actualizaciones en el ORM Eloquent ( una vez creadas las tablas en las migraciones, para modificaciones posteriores )

## Creando la tabla users con migraciones

Laravel nos ha dejado preparada ya unas migraciones básicas que incluyen lo que hemos ejecutado para tener el módulo de autenticación. Así que ahora basta con ejecutar las migraciones y nos genera la tabla en la base de datos

```
php artisan migrate
```

Ahora vamos a agregar un campo rol en la tabla usuario:

```
php artisan make:migration add_role_column_to_users
```

Abrimos el fichero de migración que nos ha creado

Contemplamos dos roles:

```
public function up()
{
    Schema::table('users', function (Blueprint $table) {
        $table->enum('role', ['user', 'admin'])->default('user');
    });
}
```

Finalmente ejecutamos el comando de las migraciones:

```
php artisan migrate
```

Ahora ya está operativa la página de registro, login, etc

## Usando el Middleware de Autenticación

En los pasos del anexo se pueden ver los ficheros que están vinculados a la autenticación. El caso es que hay vistas: registro, un login y sus correspondientes controladores.

Laravel hace uso del concepto de Middleware como ya hemos nombrado. Y el más habitual de todos es auth(). Este middleware se establece indicando que la solicitud no puede tener lugar si el usuario no está convenientemente autenticado. Así si está autenticado lo deja pasar, en otro caso lo envía al login. Veamos ejemplo para: HomeController.php Echemos un vistazo a su código:

```
class HomeController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }

    /**
     * Show the application dashboard.
     *
     * @return \Illuminate\Contracts\Support\Renderable
     */
    public function index()
    {
        return view('home');
    }
}
```

En el constructor dice:

```
public function __construct()
{
    $this->middleware('auth');
}
```

Está diciendo que para el acceso a cualquier función de esta clase debe pasar por el middleware('auth') o en otras palabras, debe estar autenticado. Este middleware te envía a login si no estás autenticado Si hubiera otro middleware ( por ejemplo que se accede únicamente si se es admin ) se pone uno después de otro:

```
public function __construct()
{
    $this->middleware('auth');
    $this->middleware('admin');
```

```
}
```

En ocasiones nuestro controlador tiene varias funciones y queremos que no filtre por autenticación las function: `getActivate()` y `getLogin()` pero sí controle el resto Eso lo hacemos así:

```
public function __construct()
{
    $this->middleware('auth')->except(['getActivate','getLogin']);
}
```

Como vemos: `except()` es un método que recibe un array con el nombre de los métodos que no deben pasar por el middleware

También podemos hacerlo al revés. Decimos las funciones que deben verse afectadas por el middleware:

```
public function __construct()
{
    $this->middleware('auth')->only(['home','share']);
}
```

Con la función: `only()` le especificamos el nombre de los métodos del controlador que deben ser filtrados por el middleware

Otra forma de establecer el middleware es cuando creamos las rutas ( por ejemplo en `web.php` ) Así, la siguiente ruta hace que la ruta: `/mostrardatosbancarios` correspondiente al método del controlador: `mostrarDatosBancarios` pase por el middleware `auth`:

```
Route::get('/mostrardatosbancarios', [
    'middleware' => 'auth',
    'uses' => 'GestionarBancoController@mostrarDatosBancarios'
]);
```

La autenticación gestionada por laravel permite acceder a la info de usuario en todo momento mediante el `helper auth()` . También tenemos `@auth` en blade que hace que lo que esté entre esa instrucción y `@endauth` se ejecuta únicamente si está autenticado

```
@auth
Usuario: {{auth()->user()->nombre}}
@endauth
```



El código anterior es de una vista Blade. @auth implica que lo que contiene esa parte de la vista se ejecuta únicamente si autenticado:

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

la parte de @guest sería para no autenticados.

Antes vimos también: auth()->user()->nombre con ese código estamos obteniendo el usuario y de éste, su nombre

Continuando con nuestra aplicación para grabar el carrito hay que estar autenticado. Nosotros ya tenemos un controlador para la gestión del carrito, en ese controlador tenemos un método para agregar producto al carrito. Esa parte no es necesario estar autenticado. Pero ahora le queremos agregar la funcionalidad de grabar el carrito ( crear el pedido ) Esa funcionalidad no debiera ser accesible a un usuario no autenticado

## Creando Middleware

Hemos usado el middleware auth pero no hemos creado nuestro propio middleware. Imaginemos que en nuestra aplicación hay una página o varias páginas reservadas a administradores ( por ejemplo, con la posibilidad de eliminar un pedido ) El acceso a esa página no se controla con auth ya que ese middleware es únicamente para usuario autenticado, nada relacionado con roles.

Vamos a crear un middleware para el roladmin y lo aplicaremos a aquellas rutas que queramos

Para crear un middleware nos apoyamos en el comando artisan:

```
php artisan make:middleware RolAdmin
```

El comando anterior creó una clase llamada RolAdmin en: **App/Middleware**

Veamos ya con contenido esa clase y lo explicamos:

```
class RolAdmin
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        $usuario = auth()->user();
        if(isset($usuario) && $usuario->role == "admin"){
            return $next($request);
        }
        abort(403); // devuelve forbidden
    }
}
```

Vemos \$next(\$request) De esta forma hacemos que la petición continúe al siguiente paso ( ya sea otro middleware, un controlador, etc ) También vemos que tomamos la información del usuario actual mediante auth() → user() Acto seguido se busca si ese usuario tiene el rol: “admin”.

Si el usuario no está autenticado o no tiene el rol admin se muestra un mensaje prohibiendo el acceso

Para que el middleware lo use laravel hay que decirle que use esa clase como un middleware en el fichero: `App\Http\Kernel.php` ( en el array: `$routeMiddleware` )

```
protected $routeMiddleware = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,  
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,  
    'can' => \Illuminate\Auth\Middleware\Authorize::class,  
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,  
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,  
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,  
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,  
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailsVerified::class,  
    'roladmin' => \App\Http\Middleware\RolAdmin::class,  
];
```

Observar que hemos agregado una etiqueta llamada: 'roladmin' que nos servirá para invocar el middleware. Ahora por ejemplo, si queremos controlar el acceso a una página: administracion pondríamos en web.php:

```
Route::get('/administracion', [AdministracionController::class, 'administrar'])  
->middleware('auth', 'roladmin');
```

Vemos que le hemos indicado que para poder acceder a la página tiene que pasar por el filtro del middleware: 'auth' y luego por el filtro del middleware: 'roladmin'

## Anexo: Sustituir la clase User por una personalizada

Con carácter general hay que tener en cuenta que las clases del modelo tienen, por defecto en laravel, campos timestamp: created\_at, updated\_at, etc que informan de la fecha de creación del registro y demás. Si eso no tiene coherencia con las tablas que tenemos en nuestra base de datos vamos a tener problemas. Podemos desactivar esos campos informando en la clase del modelo que no queremos que guarde en base de datos los timestamps ( por ejemplo si vamos a usar una tabla usuario que teníamos de antes para la autenticación y no contempla timestamps tendremos fallos si no lo desactivamos, lo mismo ocurre con rol y usuario\_rol)

```
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Sanctum\HasApiTokens;

class Usuario extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
    public $timestamps = false;

    //suponiendo un caso N:M entre tabla rol y usuario:
    public function roles()
    {
        return $this->belongsToMany(
            Rol::class,
            'usuario_rol',
            'fk_usuario',
            'fk_rol'
        );
    }
}
```

En el caso del ejemplo estamos diciendo que la tabla-clasemodelo usuario no va a registrar los timestamps

Observar que hemos hecho que extienda de Authenticatable (alias de User) esto es importante porque esta clase va a reemplazar la clase User que por defecto usa laravel en la autenticación. Si hereda de User mantiene todo el comportamiento de User y será una clase válida para el sistema de autenticación de laravel

Como hemos dicho la clase Usuario va a reemplazar a la clase User, así que tenemos que modificar unas cuantas cosas para que así sea. Sobre todo en el fichero: `config/auth.php`:

Para que sea más fácil identificar los cambios en la tabla de la base de datos se ha antepuesto: `ddbb` Así el nombre de la tabla que gestiona los usuarios es:

```
ddbbusuario( idusuario, ddbbnombre, ddbbclave )
```

donde `idusuario` es primary-key autoincremental. `ddbbnombre` es `varchar(45)` y `ddbbclave` es `varchar(145)` ( recordar que los hash pueden ser grandes )

La clase modelo para esa tabla la hemos generado con:

```
php artisan krlove:generate:model ModelUsuario --table-name=ddbbusuario --no-timestamps
```

Así que la clase que queremos que sea la que gestione los usuarios se llama: `ModelUsuario`

(observar que hemos dejado comentado lo que estaba antes y ya no va a estar -verde- y ponemos lo que debemos establecer en azul )

Se ha puesto el nombre: **providermodelusuario** como una etiqueta para que se observe que lo único relevante es que en este fichero siempre referenciamos a ese nombre de etiqueta

```
'defaults' => [
'guard' => 'web',
'passwords' => 'providermodelusuario',
/*
'passwords' => 'users',
*/
],

'guards' => [
/*
'web' => [
```

```
'driver' => 'session',
'provider' => 'users',
],
*/
'web' => [
'driver' => 'session',
'provider' => 'providermodelusuario',
],
```

```
'providers' => [
'users' => [
'driver' => 'eloquent',
'model' => App\User::class,
],

'providermodelusuario' => [
'driver' => 'eloquent',
'model' => App\ModelUsuario::class,
],
],
```

```
'passwords' => [
'users' => [
'provider' => 'users',
'table' => 'password_resets',
'expire' => 60,
'throttle' => 60,
],

'providermodelusuario' => [
'provider' => 'providermodelusuario',
'table' => 'password_resets',
'expire' => 60,
'throttle' => 60,
],
],
```

## Modificando el registro de un usuario

Implica modificar algo en la plantilla blade de registro y en el controlador correspondiente ( básicamente tiene que ser el mismo nombre de atributo que se envía y lo que se recibe )

Para hacerlo lo más sencillo posible vamos a dejar la plantilla de registro sin modificar aunque no necesitemos el correo electrónico. Tal plantilla si la queremos modificar está en:

**resources/views/auth/register.blade.php**

En el controlador ( **app/Http/Controllers/Auth/RegisteredUserController.php** ) tenemos que modificar la función `store()` ( que es la que almacena en base de datos ) para que guarde los campos de nuestro usuario. En este caso vemos ejemplo de un Usuario con únicamente nombre y password:

```
public function store(Request $request)
{
    /*
    $request->validate([
        'name' => ['required', 'string', 'max:255'],
        'email' => ['required', 'string', 'email', 'max:255', 'unique:'.User::class],
        'password' => ['required', 'confirmed', Rules\Password::defaults()],
    ]);
    */

    $user = Usuario::create([
        'nombre' => $request->name,
        //'email' => $request->email,
        'password' => Hash::make($request->password),
    ]);

    event(new Registered($user));

    Auth::login($user);

    return redirect(RouteServiceProvider::HOME);
}
```

## Modificando el login de un usuario

La plantilla blade está en: **resources/views/auth/login.blade.php**

Como ejemplo vemos que modificamos donde se introduce el correo por un campo nombre:

```
<form method="POST" action="{{ route('login') }}">
@csrf

<div>
<x-input-label for="nombre" :value="__('Nombre')"/>

<x-text-input id="nombre" class="block mt-1 w-full" type="text" name="nombre" required autofocus />

<x-input-error :messages="$errors->get('email')" class="mt-2" />
</div>

<!-- Email Address
<div>
<x-input-label for="email" :value="__('Email')"/>

<x-text-input id="email" class="block mt-1 w-full" type="email" name="email" :value="old('email')"
required autofocus />

<x-input-error :messages="$errors->get('email')" class="mt-2" />
</div>
-->
<!-- Password -->
```

Básicamente es la misma plantilla reemplazando el campo correo por uno para nombre

El controlador para login está en: **app/Http/Requests/Auth/LoginRequest.php**



Básicamente eliminamos el campo email de las validaciones ( function rules() ) y cambiamos name a nombre:

```
public function rules()
{
    return [
        //'email' => ['required', 'string', 'email'],
        'nombre' => ['required', 'string'],
        'password' => ['required', 'string'],
    ];
}
```

Y le decimos que campos son los que tiene que verificar en la base contra la base de datos ( function authenticate() )

```
public function authenticate()
{
    $this->ensureIsNotRateLimited();

    if (! Auth::attempt($this->only('nombre', 'password'), $this->boolean('remember'))) {
        RateLimiter::hit($this->throttleKey());

        throw ValidationException::withMessages([
            'email' => trans('auth.failed'),
        ]);
    }

    RateLimiter::clear($this->throttleKey());
}
```

Finalmente si también se quiere que en el dashboard se muestre correctamente el nombre de usuario hay que tener en cuenta que como nuestra clase Usuario lo que tiene es un campo: nombre en lugar de un campo name, entonces ya no llamamos a Auth:user()->name sino a Auth:user()->nombre ( fichero: **resources/views/layouts/navigation.blade.php** )

```
<!-- Settings Dropdown -->
<div class="hidden sm:flex sm:items-center sm:ml-6">
<x-dropdown align="right" width="48">
<x-slot name="trigger">
```

```
<button class="inline-flex items-center px-3 py-2 border border-transparent text-sm leading-4 font-medium rounded-md text-gray-500 bg-white hover:text-gray-700 focus:outline-none transition ease-in-out duration-150">
<div>{{ Auth::user()->nombre }}</div>
```

Juan Carlos Pérez Rodríguez