ELOQUENT - ORM

ORM (**O**bject - **R**elational - **M**apping)

El mapeo objeto-relacional es una técnica para crear un “puente” entre la programación orientada a objetos, y el acceso a las tablas de una base de datos relacional.

Mediante ORM, utilizamos programación orientada a objetos para acceder a nuestras tablas relacionales. Lo habitual es:

- Utilizar una clase para cada tabla
- Instanciar objetos de la clase para contener la información de los registros de esa tabla.
- Los métodos de la clase se encargarán de realizar las operaciones de actualización.

Eloquent

Eloquent es un mapeador objeto relacional que viene incluido por defecto en Laravel. Nos permite manejar nuestras bases de datos de forma muy sencilla.

Una de las ventajas de Eloquent, es que gestiona las relaciones de forma automática.

Si utilizamos Eloquent, no tendremos que escribir SQL, pues sus métodos se encargan de la persistencia de la información.

La documentación de Eloquent está en:

https://laravel.com/docs/11.x/eloquent

Eloquent vs DB

Para manejar nuestras bases de datos desde Laravel, podemos utilizar:

- Eloquent: Permite abstraer el acceso a la base de datos con un mapeo objeto-relacional, facilitando el desarrollo.
- Fachada DB: Este constructor de queries es muy eficiente para el manejo de grandes cantidades de información.

Nos preguntaremos entonces, ¿qué es mejor para acceder a nuestra base de datos? ¿Tengo que aprender eloquent o basta con utilizar SQL con DB?.

Ventajas de Eloquent:

- Sintaxis más sencilla que la fachada DB, por tanto más legible y fácil de mantener
- Buena solución para gestionar una entidad en modo CRUD
- Gestiona las relaciones
- Gestiona el SQL, no es necesario conocer su sintaxis concreta
- Si decido cambiar a otro gestor de base de datos, no tengo que modificar el código, ya que las diferencias de SQL las gestiona Eloquent.

En resumen, Eloquent ofrece todas las ventajas de la programación orientada a objetos

La **desventaja** de Eloquent es un peor rendimiento.

Recomendaciones:

- Se utiliza Eloquent para el desarrollo de interfaz de usuario que procese registros individuales en modo CRUD, o para joins simples con pocos registros.
- Si trabajamos con transacciones de cientos o miles de registros, es preferible utilizar DB.
- Se utiliza DB para manejo masivo de datos, tales como exportaciones, etc

Funcionamiento de Eloquent

Si utilizamos Eloquent, crearemos un modelo para cada una de las tablas de nuestra base de datos.

Un modelo es una clase, que hereda de la clase Model. La clase Model tiene muchísimos métodos, que nos permiten realizar consultas y actualizaciones de forma sencilla, sin necesidad de escribir SQL.

¿Cómo creamos el modelo MVC en Laravel?

Lo haremos mediante el comando artisan podemos crear un modelo, asociarlo a una tabla, y hacer que se construya una migración automáticamente si utilizamos el flag -m

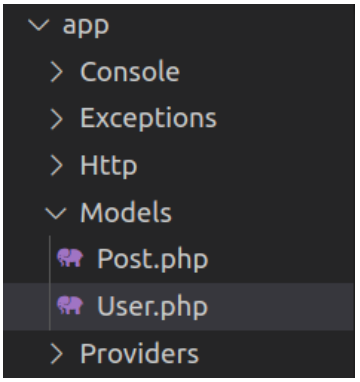
```
$ php artisan make:model Post -m
```

En nuestro caso empezaremos sin el flag -m (sin crear migración).

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:model Post

  INFO  Model [app/Models/Post.php] created successfully.
```

Los modelos en Laravel se almacenan en app/Models, creando un fichero, en nuestro ejemplo hemos creado un modelo Post para la tabla posts.



El modelo Post.php contiene por defecto el código siguiente:

Post.php

```
<?php

namespace App\Models;

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

class Post extends Model
{
    //
}
```

Como podemos ver, la clase extiende la clase Model.
Cuando crea un modelo, Laravel asume lo siguiente:

- **Laravel entenderá que la tabla se llama posts**, poniendo en plural la palabra del modelo, y en minúscula. **OJO**, el plural lo genera Laravel para palabras en inglés. Si llamamos “Mes” a mi modelo, Laravel no sabrá que la tabla se llama meses.
- **Laravel supondrá que hay una clave primaria llamada id**
- **Laravel supondrá que en la tabla hay unos campos created_at y updated_at (timestamps)**

El modelo Post.php nos permitirá definir la clase Post, con todos los métodos de acceso a la tabla posts.

En caso de que la tabla y/o la clave primaria no se llamen como asume Eloquent de Laravel, podremos añadir los siguientes atributos para indicarle a Eloquent el nombre de la tabla y de la clave primaria:

```
    protected $table = 'apli_posts'; // Si no se llama posts
    protected $primaryKey = 'posts_id'; // Si no se llama id
```

Lo anterior es imprescindible, ya que Laravel asume cierta nomenclatura, y si no la sigo no funcionará. Mediante estos atributos, podremos indicar que los nombres son otros. Esto nos permitirá utilizar otros idiomas aparte del inglés en los nombres de nuestros campos y tablas.

Eloquent asume que queremos que gestione automáticamente los timestamps, por lo que deben existir en la tabla los campos updated_at y created_at.

LEYENDO DATOS CON ELOQUENT

Vamos a utilizar rutas en web.php para probar Eloquent.

Tendremos que incluir nuestro modelo, esta nueva clase que hemos creado:

```
use App\Models\Post;
```

Método **all** de Model

La clase Model dispone de un método all(), que recupera todos los registros de una tabla. Como nuestro modelo hereda de Model, podemos utilizarlo:

web.php

```
use App\Models\Post;
Route::get('/find', function() {
    $posts = Post::all();
    foreach ($posts as $post) {
        echo $post->titulo . '<br>';
    }
})
```

1. La clase Post es el modelo para la tabla posts
2. El método estático all de la clase Post obtiene todos los registros de la tabla posts
3. Los datos de los registros se almacenan en la variable $posts, que es un array de objetos de la clase Post
4. La clase Post tiene un atributo por cada campo de la tabla, con el mismo nombre.
5. Cada objeto contendrá los valores de esos campos en el registro que representa

Método **find** de Model:

Podemos recuperar un único registro utilizando el método find de la clase Model, que recibe de parámetro el id del registro.

web.php

```
use App\Models\Post;
Route::get('/find/{id}', function($id) {
    $post = Post::find($id);
    return $post->titulo;
})
```

No debemos olvidar importar el modelo con use.

Método findOrFail de Model:

Para que no falle el find en el caso de que no exista ningún registro, utilizamos findOrFail, que retornará un 404-not found si el registro no existe.

web.php

```
use App\Models\Post;
Route::get('/find/{id}', function($id) {
    $post = Post::findOrFail($id);
    return $post->titulo;
})
```

Métodos **where, orderby, take y get** de Model

Eloquent cuenta con un constructor de queries en su modelo. Permite concatenar la query concatenando métodos: where, orderby, take, etc
Otra forma de recuperar registros, con condiciones variadas, es mediante el método where.

En Eloquent podemos encadenar métodos de la clase de la forma:

web.php

```
    Route::get('/findwhere', function() {
        $posts = Post::where('id','>','2')
                 ->orderbyDesc('titulo')
                 ->take(2)->get();
        return $posts;
    } );
```

- En el ejemplo anterior utilizamos where, el segundo parámetro es opcional, por defecto '='
- El **orderbyDesc** puede ser **orderby**, que sería ascendente.
- **Take(n)** recupera los n primeros registros obtenidos por la query.
- Con **get()** se ponen los registros en $posts en forma de array de arrays asociativos.

Nota:

➢ Si retorno el array de objetos con **return**, Laravel detectará que es un JSON, y retornará un json, indicándolo también en el content-type.
➢ Si hacemos **echo** del $posts, en vez de hacer return, el content-type será el de por defecto (html)

Método **firstOrFail**

Un método interesante, recupera un primer registro de varios, y si no hay ninguno falla generando una excepción (404 not found).

web.php

```
use App\Models\Post;
Route::get('/findmore', function() {
    $posts1 = Post::where('id', '<', 50)->firstOrFail();
    return $posts1;
})
```

En este caso, la información es retornada en formato JSON.

INSERTANDO Y ACTUALIZANDO DATOS

Para insertar registros, podemos manipularlos mediante objetos de nuestra clase modelo.

INSERTANDO O ACTUALIZANDO CON SAVE

Insertará o actualizará, la información asociada a la instancia del objeto.

Para insertar:

1. Instancio un nuevo objeto, y pongo los valores en sus atributos.
2. Finalmente hago save() (método no estático de Model)

web.php

```
Route::get('/basicinsert', function() {
    $post = new Post;
    $post->titulo = "Nueva inserción de titulo con ELoquent";
    $post->contenido = "Esto es verdaderamente guay";

    $post->save();

    return “Insertado el registro número: “ . $post->id;
})
```

El save() pone en el atributo $id el id del nuevo registro.
Nota: Si nuestra tabla no tiene los timestamps, fallará.

Para actualizar, en vez de instanciar un nuevo objeto, le asociamos un registro concreto.

En el ejemplo, busco con find un registro en la tabla, y lo asocio al objeto.

web.php

```
Route::get('/basicupdate', function() {
    $post = Post::find(1);
    $post->titulo = "NUEVO TITULO";
    $post->contenido = "Esto es verdaderamente guay";

    $post->save();
})
```

CREANDO DATOS CON CREATE

El método create() se utiliza para añadir datos de forma masiva en una tabla.
Por seguridad, Eloquent requiere que autoricemos la inserción de datos en las tablas con el método create, que recibe como parámetro un array con los campos..

Esta autorización se realiza en el modelo. En nuestro ejemplo, definimos el contenido para la variable **$fillable** que indicará qué campos podremos actualizar con create. Sólo los campos que indiquemos aquí podrán ser actualizados con create.

Post.php

```
    protected $fillable = [
        'titulo',
        'contenido'
    ];
```

Tras esta definición, podremos utilizar el modelo para crear contenido:

web.php

```
Route::get('/create', function() {
    Post::create(['titulo'=>'Método create', 'contenido'=>'Aprendiendo
muchísimo!!']);
})
```

Laravel generará el id los timestamps automáticamente.

MODIFICANDO DATOS CON UPDATE

Ya hicimos actualizaciones con el método save. Ahora vamos a ver el método update.

Para seleccionar el/los registros a modificar, utilizaremos where. Si necesitamos comprobar varias condiciones, podremos encadenar tantos “where” como haga falta, cada uno indicará el valor que debe cumplir el campo que indicamos como parámetro.

Finalmente, concatenamos el update. Esta función tiene como parámetros un array con las parejas campo-valor.

web.php

```
    Route::get('/update', function() {
        Post::where('id','>',2)->where('titulo','NUEVO TITULO')
          ->update([
              'titulo' => 'TITULO NOVÍSIMO',
              'contenido' => 'MI NUEVO CONTENIDO'
            ]);
        });
```

En el ejemplo anterior podemos ver cómo concatenar dos where al update.

- ➢ Un nuevo **where** es un and de la condición anterior, un **orWhere** sería un or.

El update puede concatenarse, y actuaría sobre los registros seleccionados en los where.

ELIMINANDO INFORMACIÓN

Método delete

Utilizaremos el método **delete**. Previamente tendremos que instanciar un objeto del modelo, y asignarle el registro que queremos eliminar. Para ello podemos utilizar la función find (vista anteriormente).

web.php

```
Route::get('/delete', function() {
    $post = Post::find(1);
    $post->delete();
})
```

Método destroy

Otra forma de eliminar, es mediante el método **destroy,** pasando como parámetro el valor de la clave primaria del registro que queremos eliminar.

web.php

```
Route::get('/delete2', function() {
    Post::destroy(3);
})
```

También podemos eliminar múltiples registros, con el método **destroy**, pasando como parámetro un array de valores de claves primarias. Laravel eliminará todos los registros correspondientes a dichas claves primarias:

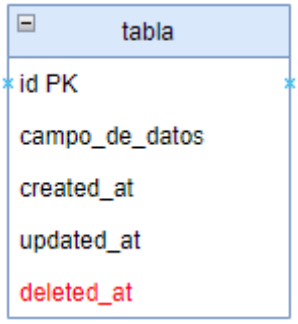
web.php

```
Route::get('/delete2', function() {
    Post::destroy([4,5]); })
```

Trash: Borrado suave

Es habitual, en el diseño de algunas tablas, hacer un borrado suave, es decir, marcar los registros como “borrados”, sin eliminarlos definitivamente.

Para implementar este diseño, se añade un campo al final de la tabla, llamado “fecha de eliminación”. Si la fecha está informada, el registro está borrado. Si la fecha no está informada, el registro está activo.



Laravel gestiona automáticamente el borrado suave de la forma:

- Añade un campo deleted_at al final de la tabla
- Si pedimos eliminar un registro, no lo borra físicamente, sino que añade fecha y hora en el campo deleted_at
- Podemos pedir la recuperación del registro, en cuyo caso dejaría en blanco el campo deleted_at
- Por defecto, cualquier búsqueda ignorará los registros con el campo deleted_at informado
- Podemos hacer búsquedas indicando que queremos ver los registros eliminados
- Podemos pedir un borrado definitivo, en cuyo caso Laravel lo elimina físicamente

Para realizar el borrado suave, tendremos que modificar el modelo, para importar la clase **SoftDeletes**.

Tendremos que añadir los campos necesarios para el borrado suave. Esto se hace con el método softDeletes.

2022_12_30_122858_create_posts_table.php

```
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('titulo');
            $table->text('contenido');
            $table->timestamps();
            $table->softDeletes();
        });
    }
```

También será necesario modificar el modelo para importar la clase SoftDeletes.

Post.php

```
class Post extends Model
{
    use SoftDeletes;
...
```

Vamos a refrescar las migraciones para que se reflejen los cambios en nuestra base de datos:

```
$ php artisan migrate:refresh
```

Obtendremos la siguiente estructura de tabla:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
1	**id**	bigint(20)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT
2	**titulo**	varchar(255)	utf8mb4_unicode_ci		No	*Ninguna*		
3	**contenido**	text	utf8mb4_unicode_ci		No	*Ninguna*		
4	**created_at**	timestamp			Sí	*NULL*		
5	**updated_at**	timestamp			Sí	*NULL*		
6	**deleted_at**	timestamp			Sí	*NULL*		

Como puede verse, se ha añadido la columna **deleted_at**. Esta columna informa que el registro ha sido eliminado, indicando fecha y hora, pero no lo elimina realmente, de forma que puede ser recuperado en cualquier momento.

Cuando insertamos información en la tabla, por ejemplo con create, la columna deleted_at estará a null por defecto.

Si mi columna existe, cuando utilice el método delete, Laravel no eliminará el registro de la tabla, sino que añadirá una fecha-hora a la columna deleted-at

Paso1: Utilizo **create** para crear tres registros iguales. Laravel creará lo siguiente:

id	titulo	contenido	created_at	updated_at	deleted_at
4	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:21	2023-01-03 13:22:21	*NULL*
5	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:23	2023-01-03 13:22:23	*NULL*
6	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:24	2023-01-03 13:22:24	*NULL*

Paso 2: Utilizo **delete** para eliminar el registro 5:

web.php

```
Route::get('/softdelete', function() {
    Post::find(5)->delete();
})
```

El resultado será el siguiente:

id	titulo	contenido	created_at	updated_at	deleted_at
4	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:21	2023-01-03 13:22:21	*NULL*
5	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:23	2023-01-03 13:26:11	2023-01-03 13:26:11
6	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:24	2023-01-03 13:22:24	*NULL*

Como puede verse, Laravel no elimina el registro, sino que le pone una fecha-hora en la columna deleted_at, para indicar que ha sido borrado… de forma suave.

A partir de ahora, esa información no será mostrada ni podrá utilizarse, aunque físicamente estará en la tabla.

Recuperando registros de la papelera

Si intento encontrar en la tabla el registro 5, que acabo de enviar a la “papelera”, Laravel no lo recuperará.

web.php

```
Route::get('/readsoftdelete', function() {
    $post = Post::find(5);
    return $post;
})
```

En el ejemplo anterior, Laravel no recupera ningún registro, porque el registro 5 tiene fecha de eliminación, es decir, se considera que está en la papelera (trash).

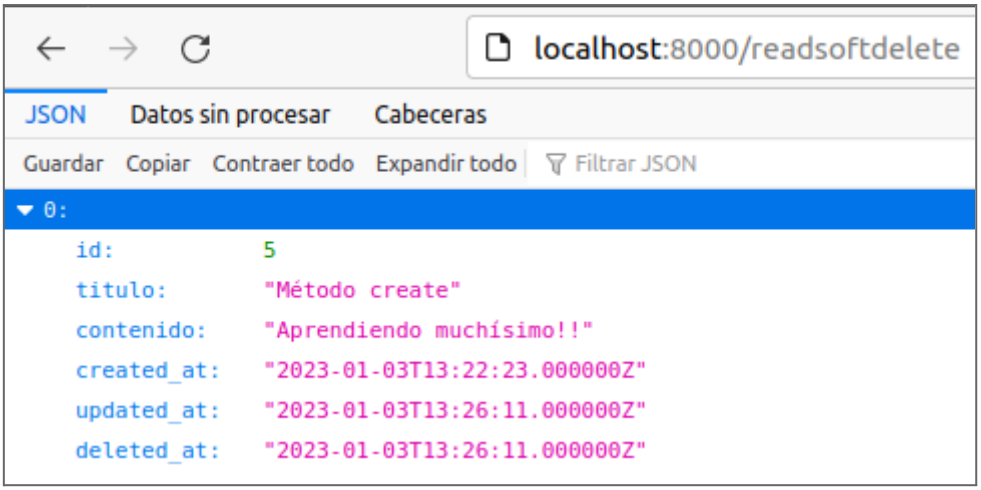
Recuperando un registro con withTrashed

¿ Qué tenemos que hacer para recuperar esta información ? Utilizaremos el método **withTrashed**:

web.php

```
Route::get('/readsoftdelete', function() {
    $post= Post::withTrashed()->where('id',5)->get();
    return $post;
})
```

Esto recuperará el registro en formato JSON:



Obtención de todos los registros: withTrashed

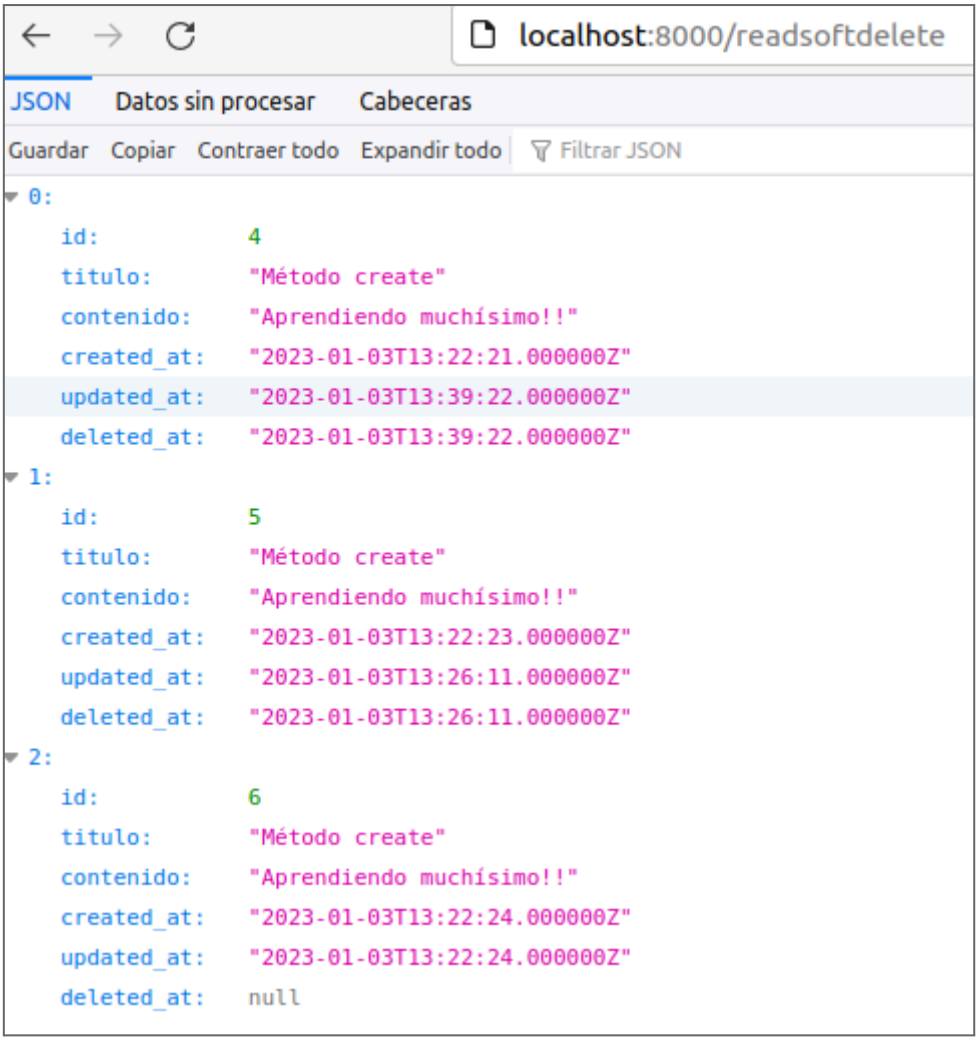
Tabla:

id	titulo	contenido	created_at	updated_at	deleted_at
4	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:21	2023-01-03 13:39:22	2023-01-03 13:39:22
5	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:23	2023-01-03 13:26:11	2023-01-03 13:26:11
6	Método create	Aprendiendo muchísimo!!	2023-01-03 13:22:24	2023-01-03 13:22:24	*NULL*

Ruta en web.php:

```
Route::get('/readsoftdelete', function() {
    $post= Post::withTrashed()->get();
    return $post;
})
```

Resultado en formato JSON:



Como puede verse, Laravel no filtra los que tienen fecha de eliminación.

Recuperación sólo de los eliminados: onlyTrashed:

Si queremos recuperar sólo los que tienen fecha de eliminación, utilizaremos el método **onlyTrashed**:

web.php

```
Route::get('/readonlytrashed', function() {
    $post = Post::onlyTrashed()->get();
    return $post;
})
```

En este caso sólo se recuperarán los registros con información en deleted_at. Si filtráramos por id con un where, podríamos obtener un registro concreto de entre los de la papelera.

Restaurando registros desde la papelera: restore

Los pasos para restaurar un registro desde la papelera, son los siguientes:

1. Con el método **withTrashed**, seleccionamos el registro encadenando un where.
2. Finalmente encadenamos un **restore**, que restaurará el registro. Al recuperarlo, lo saca de la papelera, eliminando la fecha deleted_at.
3. El registro quedará sin fecha deleted_at, es decir, ya no estará en la papelera, sino en la tabla.

web.php

```
Route::get('/restaura', function() {
    Post::withTrashed()->where('id',5)->restore();
})
```

Eliminación definitiva de un registro de la papelera: forceDelete

Al utilizar softdelete, los registros se acumulan en la papelera, puede que queramos eliminarlos de forma permanentemente de la tabla.

Para ello utilizaremos el método **forcedelete**
web.php

```
Route::get('/forcedelete', function() {
    Post::withTrashed()->where('id',4)->forceDelete();
})
```

Tenemos que tener mucho cuidado en seleccionar correctamente el registro, pues en este caso la eliminación es definitiva.