
UD10.5 – Laravel

Modelos

2º CFGS
Desarrollo de Aplicaciones Web
2022-23

1.- ORM

Un **ORM** (Object Relational Model) es una herramienta que establece una **relación directa** entre los **registros** de una tabla de la base de datos y los **objetos** de una clase del lenguaje de programación.

De esta manera los datos de la base de datos se convierten a objetos automáticamente y viceversa.

El ORM realiza estas acciones de manera automática y además incorpora herramientas para facilitar el uso de los datos en la aplicación.

1.- ORM

Laravel incorpora un ORM llamado **Eloquent**.

Para Eloquent cada tabla de la base de datos corresponde a un Modelo.

Por esta razón **lo primero** que hay que hacer para usar Eloquent en los proyectos Laravel es **crear un Modelo por cada tabla** de la base de datos del proyecto.

2.- Modelos

El comando para crear modelos es el siguiente:

```
# php artisan make:model NombreDelModelo
```

El nombre del modelo debe ser el mismo nombre que la tabla pero en singular.

El comando anterior creará en el directorio **app\Models** el archivo para el modelo.

Se puede ver que en ese directorio ya se encuentra el Modelo User.

2.- Modelos

php artisan make:model Car

```
app > Models > Car.php > ...
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Factories\HasFactory;
6  use Illuminate\Database\Eloquent\Model;
7
8  class Car extends Model
9  {
10     use HasFactory;
11 }
```

El modelo anterior correspondería a la tabla Car de la base de datos.

2.- Modelos

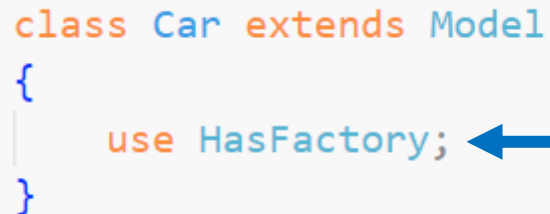
En el código generado por Laravel, dentro del modelo aparece la línea:

use HasFactory;

Esta línea sirve para poder usar las Factories.

Como se explicó en el apartado de las migraciones, las Factories permiten rellenar las tablas con valores aleatorios para usar en el desarrollo de la aplicación.

Si no se van a utilizar las Factories esa línea se puede borrar.



```
class Car extends Model
{
    use HasFactory;
}
```

3.- Consideraciones de Eloquent

En ningún momento se indica a qué tabla hace referencia el modelo, esto es porque automáticamente **Eloquent va a asociar el modelo** a la tabla que corresponda **al nombre del modelo en plural**.

De ahí la importancia de que los nombres estén en inglés.

```
app > Models > 🐘 Car.php > ...  
1  <?php  
2  
3  namespace App\Models;  
4  
5  use Illuminate\Database\Eloquent\Factories\HasFactory;  
6  use Illuminate\Database\Eloquent\Model;  
7  
8  class Car extends Model  
9  {  
10     use HasFactory;  
11 }
```

3.- Consideraciones de Eloquent

En el caso de que no se utilicen nombres en inglés o de que el nombre de la tabla no corresponda al plural del nombre del modelo se deberá indicar de la siguiente manera:

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

```
class Libro extends Model
{
    protected $table = 'Libros';
}
```


3.- Consideraciones de Eloquent

Eloquent entiende que la clave primaria **es un campo con autoincrement** llamado **id** ya que es el que automáticamente se añade a las migraciones.

Si por alguna razón la clave primaria no es un campo llamado id:

```
class Libro extends Model
{
    protected $primaryKey = 'idLibro';
}
```

3.- Consideraciones de Eloquent

Si la clave primaria **no** es un campo con autoincrement:

```
class Libro extends Model
{
    protected $incrementing = false;
}
```

Si la clave primaria **no** es un número:

```
class Libro extends Model
{
    protected $keyType = 'string';
}
```

3.- Consideraciones de Eloquent

Eloquent **no soporta claves primarias compuestas**, típicas en las tablas que se crean para relaciones con cardinalidad N:M.

Por ello se recomienda que las tablas con claves primarias compuestas tengan una clave primaria **id** y además añadir los campos que sean clave ajena.

Además, se podría indicar que los campos clave ajena sean un **campo compuesto único**:

```
$table->unique(['cliente', 'vehiculo'], 'reservas_unico');
```

Estas acciones se deben realizar en las migraciones.

3.- Consideraciones de Eloquent

Eloquent gestiona automáticamente los campos **created_at** y **updated_at** de tipo timestamp creados automáticamente.

Si no se quiere que Eloquent las gestione y se va a hacer una gestión manual se debe indicar:

```
class Libro extends Model
{
    protected $timestamps = false;
}
```

3.- Consideraciones de Eloquent

Todas las opciones vistas anteriormente se pueden combinar sin ningún problema:

```
class Libro extends Model
{
    protected $table = 'Libros';
    protected $incrementing = false;
    protected $timestamps = false;
}
```

4.- Recomendaciones con los nombres

Nombres de las migraciones: en inglés siguiendo las indicaciones previas.

Nombres de tablas: en plural y en inglés.

Nombres de los atributos de las tablas: como se prefiera ya que son para el uso del desarrollador del proyecto pero preferiblemente en inglés.

Nombres de modelos: en singular y en inglés.

Nombres de controladores: en singular, en inglés, primera en mayúscula y con la palabra Controller al final.

Rutas: en el idioma principal de la aplicación web.

Nombres de las rutas: como se prefiera ya que son para el uso del desarrollador del proyecto.

5.- Accediendo a la base de datos mediante Eloquent

Los modelos son la herramienta para trabajar con la base de datos.

Un modelo debe usarse siempre desde un controlador.

Para acceder a una tabla desde un controlador se debe indicar en ese controlador que se va a usar el modelo correspondiente añadiendo al principio una línea como la siguiente:

```
use App\Models\NombreDelModelo;
```

5.- Accediendo a la base de datos mediante Eloquent

A partir de ese momento en el **controlador** ya se puede usar el modelo para realizar las operaciones necesarias sobre la base de datos

Por ejemplo, si se quiere obtener **todos los registros de una tabla** llamada **Posts** se puede usar una de las siguientes instrucciones:

```
$coches = Post::all();
```

```
$coches = Post::get();
```

Estas instrucciones devuelven una **colección** (Collection) Eloquent que se podría decir que son **como un array de objetos de PHP**.

5.- Accediendo a la base de datos mediante Eloquent

La colección obtenida se puede enviar a una vista mediante compact.

En la vista se puede recorrer esa colección para mostrar la información.

Al ser cada elemento un objeto se usa el operador -> para acceder a las propiedades del objeto.

```
class PostController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $posts = Post::all();
        return view('posts.index', compact('posts'));
    }
}
```

```
@extends('layout')

@section('titulo', 'Listado de posts')

@section('contenido')
    <h1>Listado de posts</h1>
    @forelse ($posts as $post)
        {{$post->titulo}}
    @empty
        No hay posts.
    @endforelse
@endsection
```

Práctica

Actividad 7: Modelo Post.

6.- Ordenando los resultados

Las instrucciones vistas anteriormente recuperan todos los registros de la tabla ordenados por su id (orden en el que se guardaron).

Si se desea obtener un orden diferente se pueden utilizar las siguientes instrucciones:

orderBy

```
$posts = Post::orderBy('title')->get();  
$posts = Post::orderBy('title', 'ASC')->get();  
$posts = Post::orderBy('title', 'DESC')->get();  
$posts = Post::orderBy('title')  
                ->orderBy('created_at', 'DESC')  
                ->get();
```

orderByDesc

```
$posts = Post::orderByDesc('title')->get();  
$posts = Post::orderByDesc('visibility', 'id')->get();  
$posts = Post::orderByDesc('visibility', 'title', 'id')->get();
```

7.- Paginación automática

Eloquent ofrece un mecanismo para **paginar** los resultados de manera **automática**.

Controlador (opción 1)

Anterior, siguiente y números de página

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

Controlador (opción 2)

Anterior y siguiente

```
public function index()
{
    $posts = Post::simplePaginate(5);
    return view('posts.index', compact('posts'));
}
```

Vista

Automáticamente se añaden los enlaces a las páginas
Si no hay páginas no se mostrará ningún enlace

```
@extends('layout')

@section('titulo', 'Listado de posts')

@section('contenido')
    <h1>Listado de posts</h1>
    @forelse ($posts as $post)
        <a href="{{route('mostrar_post', $post->id)}}">{{$post->title}}</a>
        <br><br>
    @empty
        No hay posts.
    @endforelse

    {{$posts->links()}}
@endsection
```

7.- Paginación automática

En Laravel el estilo de los enlaces de paginación en las vistas lo ofrece el framework Tailwind CSS, esto hace que se necesite de CSS para visualizarlo correctamente.

Para tener una visualización más práctica hasta que se apliquen estilos CSS al proyecto se puede cambiar el estilo por el que ofrece Bootstrap.

Hay que añadir las siguientes líneas al archivo **app/Providers/AppServiceProvider.php**

Al principio del archivo:

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

En el método boot añadir una de las siguientes opciones según el diseño que se quiera:

```
Paginator::useBootstrap();  
Paginator::useBootstrapFour();  
Paginator::useBootstrapFive();
```

7.- Paginación automática

También se puede diseñar un estilo propio para los enlaces de paginación, para ello en el archivo **app/Providers/AppServiceProvider.php** se debe poner lo siguiente:

```
Paginator::defaultView( 'NombreVista' );  
Paginator::defaultSimpleView( 'NombreVistaSimple' );
```

Y a continuación crear una vista con el nombre indicado y añadir el código Blade para mostrar la paginación.

No es necesario crear añadir las dos opciones si no se necesitan.

7.- Paginación automática

En la captura se muestra el código que puede servir como base para crear un estilo de enlaces de paginación personalizado.

En el código se utiliza la variable `$paginator` la cual se encuentra en la [documentación](#) donde se pueden consultar todos los métodos de los que dispone.

También se utiliza la variable `$elements` que contiene un array con las URL de las páginas.

Las variables `$paginator` y `$elements` solo están disponibles en esta vista ya que se inyectan automáticamente desde el archivo **app/Providers/AppServiceProvider.php**.

```
@if ($paginator->hasPages())
    <nav class="text-center">
        <ul class="pagination pagination-md">
            {{-- Previous Page Link --}}
            @if ($paginator->onFirstPage())
                <li class="disabled"><span>@lang('pagination.previous')</span></li>
            @else
                <li><a href="{{ $paginator->previousPageUrl() }}" rel="prev">@lang('pagination.previous')</a></li>
            @endif

            {{-- Pagination Elements --}}
            @foreach ($elements as $element)
                {{-- "Three Dots" Separator --}}
                @if (is_string($element))
                    <li class="disabled"><span>{{ $element }}</span></li>
                @endif

                {{-- Array Of Links --}}
                @if (is_array($element))
                    @foreach ($element as $page => $url)
                        @if ($page == $paginator->currentPage())
                            <li class="active"><span>{{ $page }}</span></li>
                        @else
                            @if (in_array($paginator->currentPage(), [$page - 1, $page, $page + 1]))
                                <li><a href="{{ $paginator->url($page) }}">{{ $page }}</a></li>
                            @else
                                <li class="hidden-xs"><a href="{{ $paginator->url($page) }}">{{ $page }}</a></li>
                            @endif
                        @endif
                    @endforeach
                @endif
            @endforeach

            {{-- Next Page Link --}}
            @if ($paginator->hasMorePages())
                <li><a href="{{ $paginator->nextPageUrl($paginator) }}" rel="next">@lang('pagination.next')</a></li>
            @else
                <li class="disabled"><span>@lang('pagination.next')</span></li>
            @endif
        </ul>
    </nav>
@endif
```

Práctica

Actividad 7b:

Añade la paginación para que se vean 4 posts por página.

8.- Condiciones en los resultados

En ocasiones es necesario filtrar los registros de una tabla antes de mostrarlos.

Para esto se utiliza el método **where**:

```
public function index()
{
    $posts = Post::where('visibility', 1)->get();
    return view('posts.index', compact('posts'));
}
```

Hay que tener especial cuidado con las acciones sobre el modelo porque no todas devuelven una colección, en ese caso se debe añadir el método **get()** al final de la instrucción, como en el ejemplo.

8.- Condiciones en los resultados

Algunos ejemplos del uso del método **where**:

```
$posts = Post::where('visibility', 1)->paginate(10);

$sales = Sale::where('company', 'Nintendo')->get();

$sales = Sale::where('company', 'Nintendo')
               ->where('price', '<', 50)
               ->get();

$sales = Sale::where('company', 'Nintendo')
               ->orWhere('price', '<', 50)
               ->pagination(5);
```

9.- Métodos personalizados

En ocasiones los métodos que ofrece un CRUD no son suficientes para todas las acciones que se requieren sobre una tabla.

En esos casos será necesario crear métodos extra en el controlador y las rutas para esas acciones extra.

Cuando se añaden las rutas del CRUD al archivo de rutas y además se añaden rutas extra, estas últimas deben definirse antes que las del CRUD.

```
Route::get('posts/admin', [PostController::class, 'invisibles']);  
Route::resource('posts', PostController::class);
```

Práctica

Actividad 8: Recuperando datos con condiciones.

10.- Recuperando datos de un registro concreto

Con las acciones vistas anteriormente hemos conseguido recuperar un conjunto de registros de una tabla de la base de datos.

Es habitual también poder acceder a un solo registro.

Este acceso se suele realizar a partir de la lista mostrada anteriormente.

En la vista que muestra el listado se debe añadir un enlace a la ruta de un elemento concreto:


POST	posts	posts.store > PostController@store
GET HEAD	posts/crear	posts.create > PostController@create
GET HEAD	posts/{post} <----->	posts.show > PostController@show
PUT PATCH	posts/{post}	posts.update > PostController@update
DELETE	posts/{post}	posts.destroy > PostController@destroy
GET HEAD	posts/{post}/editar	posts.edit > PostController@edit
GET HEAD	sales	sales.index > SaleController@index
POST	sales	sales.store > SaleController@store
GET HEAD	sales/crear	sales.create > SaleController@create
GET HEAD	sales/empresa/{nombre}	ofertasempresa > SaleController@empresa
GET HEAD	sales/{sale} <----->	sales.show > SaleController@show
PUT PATCH	sales/{sale}	sales.update > SaleController@update
DELETE	sales/{sale}	sales.destroy > SaleController@destroy
GET HEAD	sales/{sale}/editar	sales.edit > SaleController@edit

10.- Recuperando datos de un registro concreto

Como en la **vista** se recibe toda la lista de elementos, cuando se muestra un elemento solo hay que crear el enlace a la ruta indicada:

```
@section('contenido')
  <h1>Listado de posts</h1>
  @forelse ($posts as $post)
    <a href="{{route('posts.show', $post->id)}}">{{$post->title}}</a>
    <br><br>
  @empty
    No hay posts.
  @endforelse

  {{$posts->links()}}
@endsection
```



10.- Recuperando datos de un registro concreto

El siguiente paso es indicar en **método show del controlador** que se obtengan los datos del registro con el **id** indicado para poder pasárselo a la vista **show**.

Para esto se utiliza el método **find()**:

```
public function show($id)
{
    $post = Post::find($id);
    return view('posts.show', compact('post'));
}
```

10.- Recuperando datos de un registro concreto

Ahora que la **vista** ya recibe un objeto concreto se pueden mostrar sus elementos:

```
@extends('layout')

@section('titulo', $post->title)

@section('contenido')
    <h1>{{$post->title}} (<a href="{{route('posts.edit', ['post' => $post->id])}}">Editar post</a></h1>
    <br>
    Fecha: {{$post->created_at}}
    <br><br>
    {{$post->content}}
@endsection
```


10.- Recuperando datos de un registro concreto

A la hora de recuperar los datos de un registro concreto se debe comprobar que se obtienen datos correctos.

Por ejemplo, hay que tener en cuenta:

- Existe un registro con el id recibido.
- No tiene algún atributo que limite su visibilidad.
- El usuario tiene permiso para ver el registro.


```
public function show($id)
{
    $post = Post::findOrFail($id);
    if($post->visibility==0)
        return redirect()->route('posts.index');
    //abort(404);
    return view('posts.show', compact('post'));
}
```

11.- Eliminación de registros de la base de datos.

Hasta este punto se ha visto la acción **read** del CRUD que es la más sencilla.

A continuación, se verá la acción **delete** que permite eliminar registros.

Estudiando las rutas que se crean para un controlador de tipo recurso, se puede observar que hay una con el método **DELETE**:



POST	posts	posts.store > PostController@store
GET HEAD	posts/crear	posts.create > PostController@create
GET HEAD	posts/{post}	posts.show > PostController@show
PUT PATCH	posts/{post}	posts.update > PostController@update
DELETE	posts/{post}	posts.destroy > PostController@destroy
GET HEAD	posts/{post}/editar	posts.edit > PostController@edit
GET HEAD	sales	sales.index > SaleController@index
POST	sales	sales.store > SaleController@store
GET HEAD	sales/crear	sales.create > SaleController@create

11.- Eliminados registros de la base de datos.

Todas las **peticiones** que no sean del tipo **GET** o **HEAD** necesitan de un **formulario** para poder ser enviadas desde el cliente correctamente.

De esa manera el servidor sabrá a qué método del controlador remitirlas.

Así, para acceder a las rutas con **POST**, **PUT** y **DELETE** se debe añadir en la página **HTML** un formulario en el que en el atributo **action** se indique la ruta destino, en el atributo **method** se debe indicar el valor **post** y dentro el formulario se añadirá una instrucción que indique el tipo de petición si la petición es **PUT** o **DELETE**.

Además, para evitar posibles peticiones desde servidores externos también se debe añadir dentro del formulario una instrucción que añadirá un **token** de seguridad.

11.- Eliminados registros de la base de datos.

De esa manera el servidor sabrá a qué método del controlador remitirlas.

Así, para acceder a las rutas con POST, PUT y DELETE:

- Se debe añadir en la página HTML un **formulario**.
- En el que en el atributo **action** se debe indicarla ruta destino.
- El método del formulario debe ser **POST**.
- Dentro del formulario una **instrucción** que indique el tipo de petición.
- Dentro del formulario una instrucción que añadirá un **token**.

11.- Eliminados registros de la base de datos.

En la **vista** posts.show:

```
@extends('layout')

@section('titulo', $post->title)

@section('contenido')
    <h1>{{ $post->title }} (<a href="{{ route('posts.edit', ['post' => $post->id]) }}">Editar post</a></h1>
    <br>
    Fecha: {{ $post->created_at }}
    <br><br>
    {{ $post->content }}
    <br><br>
    <form action="{{ route('posts.destroy', ['post' => $post->id]) }}" method="post">
        @csrf
        @method('delete')
        <input type="submit" value="Eliminar">
    </form>
@endsection
```

11.- Eliminado registros de la base de datos.

En el **método destroy** del controlador:

```
public function destroy($id)
{
    Post::findOrFail($id)->delete();
    return redirect()->route('posts.index');
}
```

Práctica

Actividad 9:

Detalle de un post y eliminación de un post.

12.- Creando modelos, migraciones y controladores.

Se ha visto que existe una **relación directa** entre los siguientes elementos:

- **Migración**
- **Modelo**
- **Controlador tipo recurso**

Por esa razón Laravel ofrece un comando que permite crear esos tres elementos de golpe:

```
# php artisan make:model Book -mcr
```

El comando anterior creará los archivos correspondientes al modelo, a la migración y al controlador de manera que estén **enlazados directamente**.

-mcr → migration controller resource

12.- Creando modelos, migraciones y controladores.


Al crear el controlador junto al modelo y a la migración, Laravel ya incluye en el controlador la **inyección de código** para el modelo.

```
# php artisan make:model Book -mcr
```

De esta manera en los métodos del controlador en los que se usa un registro de la base de datos no es necesario buscar dicho elemento ya que Laravel lo enlaza automáticamente evitando tener que hacer comprobaciones de si existe el elemento en la base de datos o no.

```
public function show(Book $book)
{
    //
}
```

```
public function show(Book $book)
{
    return view('books.show', compact('book'));
}
```



```
public function destroy(Book $book)
{
    $book->delete();
    return redirect()->route('books.index');
}
```



Práctica

Actividad 10:

Creación conjunta del modelo, migración y controlador.