

# Gestión de datos a través de Frameworks en PHP.

DESARROLLO WEB EN ENTORNO SERVIDOR

GARRIDO PORTES, GUILLERMO



Atribución-NoComercial-SinDerivadas  
4.0 Internacional (CC BY-NC-ND 4.0)

# Gestión de datos a través de Frameworks en PHP.

---

|   |    |
|---|----|
| Gestión de datos a través de Frameworks en PHP.....       | 1  |
| 1. Bases de datos desde Laravel .....                     | 2  |
| Configurar y conectar con la base de datos.....           | 2  |
| Migraciones .....   | 3  |
| 2. El ORM Eloquent de Laravel. ....                       | 7  |
| Modelos.....  | 7  |
| Relaciones con Eloquent ORM.....                          | 9  |
| Uno a uno (OTO) .....                                     | 9  |
| Uno a muchos (OTM) .....                                  | 10 |
| Tienen uno (HOT) y Tiene muchos (HMT).....                | 11 |
| Muchos a muchos (MTM) .....                               | 12 |
| Relaciones polimórficas.....                              | 13 |
| 3. Realizar consultas a la base de datos .....            | 15 |
| Generador de consultas .....                              | 16 |
| Eloquent y las colecciones.....                           | 16 |
| Crear registros en la base de datos .....                 | 17 |
| Asignación masiva .....                                   | 18 |
| Seeders.....  | 19 |
| <i>Model Factories</i> .....                              | 21 |
| Leer y listar registros de la base de datos .....         | 24 |
| Actualizar y eliminar registros de la base de datos ..... | 25 |
| 4. Formularios .....                                      | 26 |
| Validar formularios.....                                  | 26 |
| Mostrar errores de validación de formularios.....         | 29 |
| Cargar y recargar información en los formularios.....     | 29 |

# 1. Bases de datos desde Laravel

Laravel simplifica la interacción con las bases de datos pudiendo configurar una gran variedad de bases de datos. Para ello proporciona diferentes herramientas como el uso de **RAW SQL** (consultas SQL sin procesar), utilizando el *QueryBuilder* (que es un generador de consultas fluido) o mediante el uso de su sistema de Mapeo Objeto-Relacional (ORM) **Eloquent**.

## Configurar y conectar con la base de datos

Para poder configurar las bases de datos, se necesitan dos archivos principalmente. El archivo de configuración por defecto “**database.php**” que está dentro del directorio “**config**” y donde Laravel proporciona en el archivo “**database.php**” soporte para cinco bases de datos MariaDB 10.3+, MySQL 5.7+, PostgreSQL 10.0+, SQLite 3.8.8+, SQL Server 2017+.

```

1 'default' => env('DB_CONNECTION', 'mysql'),
2
3 'connections' => [
4     'mysql' => [
5         'driver' => 'mysql',
6         'url' => env('DATABASE_URL'),
7         'host' => env('DB_HOST', '127.0.0.1'),
8         'port' => env('DB_PORT', '3306'),
9         'database' => env('DB_DATABASE', 'forge'),
10        'username' => env('DB_USERNAME', 'forge'),
11        'password' => env('DB_PASSWORD', ''),
12        'unix_socket' => env('DB_SOCKET', ''),
13        'charset' => 'utf8mb4',
14        'collation' => 'utf8mb4_unicode_ci',
15        'prefix' => '',
16        'prefix_indexes' => true,
17        'strict' => true,
18        'engine' => null,
19        'options' => extension_loaded('pdo_mysql') ? array_filter([
20            PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
21        ]) : [],
22    ],

```

La configuración de las conexiones utiliza la función **env()** para buscar dentro del archivo de variables de entorno “**.env**” los valores de las diferentes variables necesarias para la configuración del sistema. Como primer parámetro buscará el nombre de la variable de entorno y si no la encuentra utilizará el predeterminado que le llega como segundo parámetro.

```

1 DB_CONNECTION=mysql
2 DB_HOST=127.0.0.1
3 DB_PORT=3306
4 DB_DATABASE=ud6_laravel_ggarrido
5 DB_USERNAME=root
6 DB_PASSWORD=

```

En caso de utilizar una base de datos SQL, se deberá crear la Base de datos para poder crear la conexión con esta.

Pero es posible añadir la configuración de otras bases de datos como MongoDB. Para ello en primer lugar se deberá descargar utilizando el gestor de paquetes **Composer** la librería de **Mongodb** para Laravel

```
proyecto-ggarrido-laravel_mongodb> composer require jenssegers/mongodb
```

A continuación, se deberá añadir dentro del array de conexiones (*connections*) en **“database.php”** la configuración para la conexión con dicha base de datos.

```
1 'connections' => [
2     'mongodb' => [
3         'driver' => 'mongodb',
4         'host' => env('DB_HOST', '127.0.0.1'),
5         'port' => env('DB_PORT', 27017),
6         'database' => env('DB_DATABASE', 'default'),
7         'username' => env('DB_USERNAME', ''),
8         'password' => env('DB_PASSWORD', '')
9     ],

```

Aunque la configuración para trabajar en local está establecida dentro del array ***connections***, es muy recomendable siempre hacer uso , aunque sean los mismos valores, de las variables de entorno en el archivo de configuración del archivo **“.env”** y estableciendo el valor de la variable **“DB\_CONNECTION”** como **“mongodb”**.

```
1 DB_CONNECTION=mongodb
2 DB_HOST=127.0.0.1
3 DB_PORT=27017
4 DB_DATABASE=ud6_laravel_ggarrido
5 DB_USERNAME=
6 DB_PASSWORD=
```

## Migraciones

Las migraciones son un elemento de Laravel que permite crear y modificar las tablas de las bases de datos.

Las migraciones contienen lo métodos donde se genera el esquema de la tabla de la base de datos y donde se destruye.

En concreto, el método **up()** define la creación o modificación del esquema de la tabla mientras que el método **down()** realizara lo opuesto, eliminando o deshaciendo la modificación de la tabla.

Además, si se utiliza una base de datos relacional se podrán establecer las relaciones de las tablas. Por ello es muy importante ver el orden de creación de estas.

Para crear una migración se puede hacer creando el archivo dentro de la carpeta *migrations* o bien utilizando el comando de “**php artisan make:migration**” seguido del nombre de la tabla.

El nombre de la migración, por convenio, suele empezar por la fecha y hora de creación siguiendo el patrón “*yyyy\_mm\_dd\_hhmmss*” seguido por la palabra *create* o *update\_to* según lo que realice, y el **nombre de la tabla** para DB al que le sigue por último la palabra *table*. Todas estas separadas por el símbolo barra baja “\_” en lugar de espacios en blanco.

En caso de que el nombre de la tabla sean palabras compuestas, también se utilizará la barra baja “\_” para separar cada palabra y siempre referenciarán sus elementos en plural.

Si se genera la migración desde el comando “*artisan*” no será necesario indicar en el nombre de la migración la fecha y hora de esta, ya que el comando lo añadirá automáticamente.

```
proyecto-ggarrido-laravel> php artisan make:migration create_customers_table
```

Dentro del esquema de la tabla autogenerado por “*artisan*”, se puede ver que cuando encuentra la palabra *create* en el nombre de la tabla genera el esquema “**Schema::create**” con los campos “**id**” y “**timestamps**”. Este último, realmente genera dos columnas en la base de datos que se utilizan para almacenar la fecha y hora de la creación “**created\_at**” y actualización “**updated\_at**” de cada registro.

```
1 <?php
2 //Nombre 2023_01_11_154311_create_customers_table.php
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6 return new class extends Migration
7 {
8     /**
9      * Run the migrations.
10     * @return void
11     */
12    public function up()    {
13        Schema::create('customers', function (Blueprint $table) {
14            $table->id();
15            $table->timestamps();
16            $table->softDeletes();
17        });
18    }
19    /**
20     * Reverse the migrations.
21     * @return void
22     */
23    public function down()    {
24        Schema::dropIfExists('customers');
25    }
26};
```

Cuando se crean las migraciones se puede proporcionar al esquema de la misma la columna “**delete\_at**” añadiendo la llamada a la función **softDeletes()**.

Esto permite que al eliminar un registro realmente no se borre de la base de datos, sino que se añade un dato de tipo “**timestamp**” de la fecha y hora que fue borrado. De este modo se puede seguir manteniendo un histórico de los datos introducidos en la base de datos.

Cuando Laravel lista los datos desde la base de datos obviara todos aquellos registros que contengan valor en la columna “**delete\_at**”.

Por otro lado, si la migración es un “**update\_to**”, la migración no debe crear un esquema, sino que se utilizara para añadir los campos que se le indiquen al esquema de creación.

```
 proyecto-ggarrido-laravel> php artisan make:migration update_to_customers_table
```

En este caso se añadirá el tipo de dato y el nombre de la columna. En caso de querer situarla detrás de una columna existente específica, se podrá hacer uso de la función **after()** pasando como parámetro la columna que la precederá.

```

1 <?php
2 //Nombre 2023_01_11_154554_update_to_customers_table.php
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6 return new class extends Migration{
7     /**
8      * Run the migrations.
9      * @return void
10     */
11    public function up(){
12        Schema::table('customers', function (Blueprint $table) {
13            $table->after('id', function ($table) {
14                $table->string('nombre');
15                $table->integer('edad');
16            });
17            $table->string('email')->unique();
18        });
19    }
20    /**
21     * Reverse the migrations.
22     * @return void
23     */
24    public function down(){
25        Schema::table('customers', function (Blueprint $table) {
26            $table->dropColumn(['nombre', 'edad', 'email']);
27        });
28    }
29 };

```



## ENLACE

En el enlace “Tipos de columnas de las migraciones” se pueden consultar los tipos de columnas que se pueden generar dentro del esquema.

<https://laravel.com/docs/migrations#available-column-types>

Una vez se han creado las migraciones se pueden crear las tablas en la base de datos utilizando el comando de “**php artisan migrate**”.

Este comando generara todas las tablas correspondientes a cada migración `create` y las modificaciones del “`update_to`”. Además, generara una tabla llamada *migrations* para llevar un control de las migraciones ya ejecutadas para evitar ejecutarlas nuevamente ya que esto eliminaría los datos de las tablas.

La tabla *migrations*, cuenta con la columna “*batch*” que proporciona un control de en qué ejecución del comando “**php artisan migrate**” se ha añadido cada migración. De este modo, se puede tener un registro de como ha ido cambiando la base de datos (algo así como un control de versiones). Este sistema también permite eliminar las ultimas migraciones realizadas (afectando únicamente al último *batch*) utilizando el comando de “**php artisan migrate:rollback**”.

```
projeto-ggarrido-laravel> php artisan migrate
INFO  Preparing database.

Creating migration table ..... 15ms DONE

INFO  Running migrations.

2019_12_14_000001_create_personal_access_tokens_table ..... 40ms DONE
2023_01_11_154311_create_customers_table ..... 7ms DONE

projeto-ggarrido-laravel> php artisan migrate
INFO  Running migrations.

2023_01_11_154554_update_to_customers_table ..... 2ms DONE
```

| id | migration   | batch |
|----|---|-------|
| 1  | 2019_12_14_000001_create_personal_access_tokens_ta... | 1     |
| 2  | 2023_01_11_154311_create_customers_table              | 1     |
| 3  | 2023_01_11_154554_update_to_customers_table           | 2     |

```
 proyecto-ggarrido-laravel> php artisan migrate:rollback
 INFO Rolling back migrations.

 2023_01_11_154554_update_to_customers_table ..... 4ms DONE
```

| <b>id</b> | <b>migration</b>                                      | <b>batch</b> |
|-----------|---|--------------|
| 1         | 2019_12_14_000001_create_personal_access_tokens_ta... | 1            |
| 2         | 2023_01_11_154311_create_customers_table              | 1            |

Si se desean añadir todas las migraciones en un único bloque de modo que queden dentro de un mismo (*batch*) eliminando el contenido de todas las tablas y volviendo a generarlo, se podrá hacer uso del comando de “`artisan migrate:fresh`”.

```
 proyecto-ggarrido-laravel> php artisan migrate:refresh
 INFO Rolling back migrations.

 2023_01_11_154311_create_customers_table ..... 7ms DONE
 2019_12_14_000001_create_personal_access_tokens_table ..... 5ms DONE

 INFO Running migrations.

 2019_12_14_000001_create_personal_access_tokens_table ..... 32ms DONE
 2023_01_11_154311_create_customers_table ..... 7ms DONE
 2023_01_11_154554_update_to_customers_table ..... 0ms DONE
```

## 2. El ORM Eloquent de Laravel.

Otro método de realizar operaciones con la base de datos es el uso del ORM de Eloquent que proporciona el *framework*.

Este método permite centralizar las peticiones referentes a cada tabla en una clase distinta llamada **modelo**, pudiendo hacer uso de los objetos de este para realizar las operaciones con la base de datos.

### Modelos

En Eloquent se utiliza un modelo para cada tabla de la base de datos excepto para las tablas pivot (las tablas que se forman por la relación muchos a muchos) que solo se crean modelos de estas en casos muy concretos.

Cuando se genera el modelo Laravel asume que este tendrá el mismo nombre que la tabla, pero en singular (ya que un objeto del modelo referenciaría a un registro de la BD) y empezando por mayúscula (como en cualquier clase PHP).

Además en caso que sean palabras compuestas se pondrá el inicio de cada palabra en mayúsculas sin espacios utilizando el estándar “**PascalCase**”.



## IMPORTANTE

Laravel tiene un sistema automático de referencia de nombres y claves. Para que este sistema funcione correctamente y pueda convertir nombres del singular al plural y viceversa, entre otras cosas, Laravel necesita que los nombres de los modelos, migraciones, atributos, etcétera sean en inglés.

Para generar un modelo se puede hacer de forma manual dentro del directorio “`app/models`” o bien haciendo uso del comando “`php artisan make:model`” seguido del nombre del modelo.

```
proyecto-ggarrido-laravel> php artisan make:model Customer
```

También es posible crear el modelo y la migración de tipo *create* a la vez si después del nombre del modelo ponemos la opción “`-m`”.

```
proyecto-ggarrido-laravel> php artisan make:model Address -m
INFO Model [proyecto-ggarrido-laravel\app\Models\Address.php] created successfully.
INFO Migration [proyecto-ggarrido-laravel_mysql\database\migrations\2023_01_11_161906_create_addresses_table.php] created successfully.
```

El modelo será una clase que extenderá de la clase *Model* de Eloquent de modo que tendrá ciertos métodos que permitirán interactuar al modelo con la base de datos.

Por otro lado, también cuenta con algunos atributos que pueden modificarse permitiendo variar el comportamiento del modelo. Por ejemplo, si se desea utilizar un modelo con nombre diferente a la tabla se puede modificar el atributo **`protected $table`**.

También será necesario utilizar el *trait* para poder hacer uso del softDelete en dicho modelo añadiendo dentro de la clase la instrucción “**`use SoftDeletes;`**” y importándolo con “**`use Illuminate\Database\Eloquent\softDeletes;`**” siempre que este configurado en su migración.

```
1 <?php
2 namespace App\Models;
3 use Illuminate\Database\Eloquent\Model;
4 /* En caso de utilizar Mongodb se debe
   sobreescribir
5    use Illuminate\Database\Eloquent\Model; por
6    use Jenssegers\Mongodb\Eloquent\Model;
7 */
8 use Illuminate\Database\Eloquent\softDeletes;
9 class Customer extends Model
10 {
11    use SoftDeletes;
12 }
```

## Relaciones con Eloquent ORM.

El ORM de Eloquent permite, a nivel de código, establecer relaciones entre los diferentes modelos. Esto permite simular las relaciones dentro de la base de datos y de este modo obtener registros según las relaciones.

Las relaciones permitirán obtener objetos de un modelo utilizando otro modelo. Por ejemplo, si se desea obtener un modelo B que se relaciona con un modelo A, se deberá crear un método interno en el modelo A con el nombre del modelo B o una variante de este.

Este método contendrá el tipo de relación entre el modelo A y el modelo B.

Del mismo modo, si se desea poder obtener un objeto del modelo A desde el modelo B, el modelo B deberá implementar un método que tenga la relación con el modelo A.

Para poder definir estas relaciones existen diferentes métodos de Eloquent que permiten establecer la relación de un modelo con el otro, según el tipo de relación que se desee crear:

### Uno a uno (OTO)

Para crear una relación OTO, se debe conocer cuál es el modelo principal y cuál es el modelo que contendrá la clave ajena.

Por ejemplo, si existen los modelos *Customer* y *Address*, y se desea obtener una dirección que pertenece a un usuario, se deberá crear un método interno en el modelo *Customer* con el nombre **address()** o una variante de este.

En este, se llamará al método de la clase *Model* de Eloquent llamado **hasOne()** que recibirá como primer parámetro el nombre del modelo, en este caso *Address*.

Como segundo parámetro se define la clave ajena que se encontrará en el modelo *Address*, siendo el tercer parámetro la clave primaria del *Customer*.

En caso de no querer establecer la clave primaria y la clave ajena, pueden omitirse estos argumentos ya que Eloquent identifica la clave ajena basándose en el nombre del modelo. De este modo en el método **address()** del modelo *Customer*, buscará la columna “id” como clave primaria y la columna “usuario\_id” como clave ajena dentro del modelo *Address*.

```

1 class Customer extends Model
2 {
3     use SoftDeletes;
4     /**
5      * Obten la dirección asociada con el cliente.
6      */
7     public function address()
8     {
9         return $this->hasOne(Address::class);
10    }
11 }
```

La relación inversa se realizaría en el modelo *Address* donde se implementará un método llamado **customer()** en el que se utilizará un método de la clase *Model* de Eloquent llamado **belongsTo()** que recibirá como primer parámetro una instancia del modelo *Customer*.

El segundo y tercer parámetro serán opcionales y en el mismo orden que como en la relación **hasOne()**.

```

1 class Address extends Model
2 {
3     /**
4      * Obten el cliente al que pertenece la dirección.
5      */
6     public function customer()
7     {
8         return $this->belongsTo(Customer::class);
9     }
10 }
```

## Uno a muchos (OTM)

Para crear una relación OTM, es muy similar a la relación OTO, pero en este caso la función del modelo principal tendrá una función que devolverá múltiples instancias del modelo que contiene la clave ajena.

Por ejemplo, en muchos países para indicar la calle se utilizan diferentes líneas de texto, si existen los modelos *Address* y *AddressLine*, y se desean obtener las líneas de una dirección, se deberá crear un método interno en el modelo *Address* con el nombre **addressLines()** o una variante de este.



### IMPORTANTE

Para poder ejecutar la relación con *AddressLine* se debe haber creado el modelo y la migración de este “`php artisan make:model AddressLine -m`”.

Además el esquema de la migración debe contar con **`$table->string('line');`** para almacenar el texto de la calle.

En este, se llamará al método de la clase “Model” de Eloquent llamado **hasMany()** que recibirá como primer parámetro el nombre del modelo, en este caso *AddressLine*.

El segundo y tercer parámetro son los mismos que en la relación **hasOne()**.

```

1 class Address extends Model{
2     ...
3     /**
4      * Obten las líneas asociadas a la dirección.
5      */
6     public function addressLines()    {
7         return $this->hasMany(AddressLines::class);
8     }
9 }
```

La relación inversa se realizaría en el modelo *AddresLine* donde se implementará un método llamado **address()** en el que se utilizará el mismo método de la clase “Model” de Eloquent **belongsTo()** que en este caso recibirá como primer parámetro una instancia del modelo *Address*.

```

1 class AddressLine extends Model
2 {
3     /**
4      * Obten la dirección a la que pertenece la línea.
5      */
6     public function address()
7     {
8         return $this->belongsTo(Address::class);
9     }
10 }
```



### IMPORTANTE

También existe la posibilidad de concatenar al método de Eloquent **hasOne()** otros métodos como **latestOfMany()**, **oldestOfMany()** o **ofMany()** que permitirán obtener de una consulta OTM solo el ultimo valor, el mas viejo o el que cumpla con la condición de tener el valor mínimo o máximo de una determinada columna.

## Tienen uno (HOT) y Tiene muchos (HMT)

La relación HOT define una relación uno a uno con otro modelo a través de un modelo intermedio. Mientras que la relación HMT define una relación uno a muchos con otro modelo a través de un modelo intermedio.

Por ejemplo, si se quieren obtener las líneas de dirección de una dirección que pertenece a un cliente, se debería contar con los modelos *Customer*, *Address* y *AddresLine*. Para establecer la relación se deberá crear un método interno en el modelo *Customer* con el nombre **addressLines()** o una variante de este.

En este, se llamará al método de la clase *Model* de Eloquent llamado **hasManyThrough()** (ya que puede haber más de una línea de dirección de que pertenezca a una dirección del cliente) que recibirá como primer parámetro el nombre del modelo que se desea obtener, en este caso *AddresLine* y como segundo parámetro el nombre del modelo que hace de intermediario, en este caso *Address*.

Después podrá contar con las claves primarias y ajena de los diferentes modelos. Para ello se le pasará como tercer parámetro la clave ajena en *Address* que será “**customer\_id**”, como cuarto parámetro la clave ajena en *AddresLine* que será “**address\_id**” y como quinto y sexto parámetros las claves primarias del modelo *Customer* y *Address* respectivamente, estas habitualmente serán “**id**” en ambos casos.

```

1 class Customer extends Model
2 {
3     ...
4     /**
5      * Obten las líneas asociadas a la dirección asociada con el cliente.
6      */
7     public function addressLines()
8     {
9         return $this->hasManyThrough(AddressLines::class, Address::class);
10    }
11 }

```

## Muchos a muchos (MTM)

Para crear una relación MTM en la base de datos es necesario contar con una tabla pivote. La tabla pivote contendrá, entre sus posibles columnas, la clave primaria de cada modelo que la forma como campos obligatorios.

Por ejemplo, si se desean almacenar las redes sociales con las que cuentan los clientes de la base de datos, se necesitarán los modelos *Customer* y *SocialNetwork*, se deberá crear un método interno en el modelo *Customer* con el nombre **socialNetworks()** o una variante de este.

En este, se llamará al método de la clase *Model* de Eloquent llamado **belongsToMany()** que recibirá como primer parámetro el nombre del modelo, en este caso *SocialNetwork*.

El segundo parámetro recibirá el nombre de la tabla pivote que en este caso debería ser “*customer\_social\_network*” ya que se debería componer por el nombre compuesto de los dos modelos que se referencian entre sí en orden alfabético siendo el tercer la clave primaria del modelo en el que se está definiendo el método “*customer\_id*” y el cuarto parámetro la clave primaria del modelo con el que se relaciona “*social\_network\_id*”.

La relación inversa devolvería todos los clientes que utilizan una red social. Para ello se implementará un método llamado **customers()** en el modelo *SocialNetwork* en el que se utilizará el mismo método **belongsToMany()** que recibirá como primer parámetro el nombre del modelo, en este caso *Customer*.

Si dentro de la tabla pivote, además de las columnas referentes a las claves primarias del modelo que la forman existen columnas adicionales, es posible utilizar después de la llamada al método **belongsToMany()** una llamada concatenada con el símbolo “*->*” al método **withPivot()** donde se le pararan las diferentes columnas separadas por comas.

```

1 class Customer extends Model
2 {
3     ...
4     /**
5      * Obten las redes sociales asociadas con el cliente.
6      */
7     public function socialNetworks()
8     {
9         return $this->belongsToMany(SocialNetwork::class)->withPivot('followers');
10    }
11 }

```

También es posible añadir las columnas *timestamps* de forma automática concatenando la llamada al método **withTimestamps()**.

```

1 class SocialNetwork extends Model
2 {
3     /**
4      * Los clientes que usan la red social.
5      */
6     public function customers()
7     {
8         return $this->belongsToMany(Customer::class)->withTimestamps();
9     }
10 }
```

Aunque no es común, en ocasiones es posible utilizar modelos para referenciar las tablas pivot. En este caso se podría referenciar la tabla pivot utilizando el modelo que la representa. Para ello se realizaría una llamada concatenada al método **using()** que recibirá como parámetro el nombre del modelo que debe usar.

```
1 return $this->belongsToMany(SocialNetwork::class)->using(CustomerSocialNetwork::class);
```



## IMPORTANTE

Para poder ejecutar la relación MTM con *SocialNetwork* se debe haber creado el modelo y la migración de este “`php artisan make:model SocialNetwork -m`”.

Además el esquema de la migración debe contar con `$table->string('nombre');` para almacenar el nombre de la red social.

Otro componente necesario es la migración de la tabla pivot `customer_social_network` “`php artisan make:migration create_customer_social_network_table`”. Además esta debe contar con:

- `$table->integer('customer_id');`
- `$table->integer('social_network_id');`
- `$table->integer('followers');`

## Relaciones polimórficas

En ocasiones encontramos modelos que pertenecen a otros modelos teniendo que implementar como clave ajena las claves primarias de estos. Puesto que la pertenencia solo será a uno de los modelos a la vez, solo se rellenará una de las claves ajenas teniendo que ser nulos el resto de ellas.

Si solo existen un par de dependencias puede ser algo poco importante, pero si este modelo pertenece a muchos modelos, el número de columnas puede aumentar haciendo poco

mantenible la base de datos, ya tendrá muchas columnas como nulas y solo rellenará la columna de la clave ajena a la que pertenezca.

Para evitar este exceso de columnas aparecieron las relaciones polimórficas.

Una relación polimórfica permite que el modelo pueda pertenecer a más de un tipo de modelo usando una sola asociación.

Para ello en lugar de establecer una columna para cada clave ajena, establece únicamente dos columnas que se compondrán por el nombre del modelo con el sufijo “**able**” y seguido de “**\_id**” y “**\_type**” respectivamente.

Por ejemplo, si se desea utilizar el modelo */image* para poder almacenar la imagen que pueda pertenecer al perfil del cliente pero que también las imágenes que puedan pertenecer a las fotos de una determinada dirección, se deberían crear en la tabla *images* las columnas “**imageable\_id**” y “**imageable\_type**”.

El comportamiento de la relación del modelo será el de almacenar en la base de datos el id del modelo al que pertenece la imagen dentro de la columna “**imageable\_id**” y el nombre del modelo padre que la crea, en la columna “**imageable\_type**”.

Los posibles métodos de Eloquent para relaciones polimórficas son:

- **morphOne()** para una relación OTO polimórfica de un único registro
- **morphMany()** para una relación OTM polimórfica de múltiples registros
- **morphTo()** para la función dentro del modelo polimórfico que solo pueda pertenecer a un único modelo al mismo tiempo. Esta se relaciona con modelos que cuentan con las funciones **morphOne()** o **morphMany()**.
- **morphToMany()** para una relación MTM polimórfica de múltiples registros desde diferentes modelos.
- **morphedByMany()** para la función dentro del modelo polimórfico que pueda pertenecer a más de un modelo al mismo tiempo. Esta se relaciona con modelos que cuentan con la función **morphToMany()**.



### IMPORTANTE

En los métodos de Eloquent **belongsTo()**, **hasOne()**, **hasOneThrough()** y **morphOne()** le permiten definir un modelo predeterminado que se devolverá si la relación dada es nula.

Para ello se añadirá después de la llamada al método la instrucción “**->withDefault()**” donde dentro del paréntesis podrá recibir un array asociativo con los atributos del modelo que se espera obtener y los valores predeterminados para estos atributos.

```

1 class Image extends Model
2 {
3     /**
4      * Obtiene el modelo propietario de la imagen.
5      */
6     public function imageable()
7     {
8         return $this->morphTo();
9     }
10 }
11
12 class Post extends Model
13 {
14     /**
15      * Obtiene la imagen del perfil del cliente.
16      */
17     public function image()
18     {
19         return $this->morphOne(Image::class, 'imageable');
20     }
21 }
22
23 class User extends Model
24 {
25     /**
26      * Obtiene las imágenes de la dirección.
27      */
28     public function image()
29     {
30         return $this->morphMany(Image::class, 'imageable');
31     }
32 }

```

### 3. Realizar consultas a la base de datos

Una vez se disponen de todos los elementos del modelo MVC, se debe separar la lógica de cada parte de la aplicación en el elemento que este destinado para ello.

De este modo, los modelos tendrán todos los métodos que permiten almacenar, obtener y eliminar la información de la base de datos, así como las relaciones que facilitan la obtención de dicha información.

Por otra parte, los controladores serán los destinados a crear la lógica necesaria de la aplicación, proporcionando los datos que requiere el usuario.

Para poder obtener la información, los controladores pueden utilizar modelos o el generador de consultas de Laravel desde alguno de sus métodos internos.

El controlador por defecto que mas se utiliza para llevar un control de la lógica de una tabla es y así poder realizar el **CRUD** (*create, read, update y delete*) de los elementos de esta es el **controlador de recursos** visto en el tema 5.

## Generador de consultas

El generador de consultas o *QueryBuilder* (QB) de Laravel permite realizar la mayoría de las operaciones utilizando el enlace de parámetros PDO para proteger la aplicación contra ataques de inyección SQL.

Para realizar las consultas con el QB es necesario hacer uso del *Facade DB* el cual contiene el método **table()** que recibe como parámetro el nombre de la tabla de la base de datos (o su alias).



### IMPORTANTE

Las *Facades* (o Fachadas) son un patrón de diseño de la Programación Orientada a Objetos que nos permite ocultar un subsistema detrás de un objeto. Proporciona una interfaz estática para poder acceder a los métodos de clases que están vinculados en el contenedor de servicios usando un método mágico.



### ENLACE

En el enlace “¿Para qué sirven las *Facades* en Laravel?” muestra un caso real de cómo se pueden utilizar los *facades*. <https://www.laraveltip.com/para-que-sirven-las-facades-en-laravel-explicadas-con-un-caso-real/>

El método `table` devolverá una consulta a la base de datos, pudiendo concatenar restricciones, pero para poder recoger una colección de los registros de esta será necesario llamar al método **get()** que ejecute la consulta.

```
DB::table('customers')->get();
```

Te pueden crear diferentes consultas haciendo uso de operaciones con métodos como **select()**, **distinct()**, **exists()**, **join()**, **unión()**, etcétera, o bien realizar consultas complejas con el método **raw()** y sus derivados.

## Eloquent y las colecciones

Todos los métodos de Eloquent que devuelvan más de un resultado de modelo devolverán una colección de registros, incluidos los resultados obtenidos a través del método **get()** o a los que se accede a través de una relación.

Las colecciones de Laravel son un tipo de clases que proporcionan una variedad de métodos para mapear y reducir la cantidad de datos. En estas, cada resultado es una instancia de un modelo.

Algunos de los métodos interesantes para establecer son:

- **first()** permite obtener el primer registro de la consulta.
- **all()** permite obtener una colección de registros.
- **where()** permite filtrar el contenido de la consulta según se cumple la condición recibida. Esta a su vez cuenta con variantes importantes como **orWhere()**, **whereNot()**, **whereBetween()**, **whereIn()**, y las negativas de las dos anteriores.
- **value()** permite recibir únicamente la columna que se le indique de un registro
- **pluck()** permite recibir las columnas que se le indiquen de una colección de registros
- **find()** busca el elemento por **id**
- **findOrFail()** muestra una página 404 en caso de no existir el elemento buscado
- **orderBy()** permite ordenar por la columna que se le indique los registros de la colección.
- **chunk()** permite filtrar el número de registros de la colección por el valor recibido como primer parámetro.
- **lazy()** permite realizar lo mismo que **chunk()** pero en lugar de tener que realizar una llamada a la base de datos para recibir la siguiente colección de registros, en este caso recibe todos los registros y los filtra dentro de un tipo de colección especial llamado *LazyCollection*.
- Las funciones de agregación como **count()**, **avg()**, **max()**, **min()** o **sum()**.



### ENLACE

En el enlace “Tipos de métodos de las colecciones” se pueden consultar los tipos de métodos que se pueden utilizar para el uso de colecciones de Laravel.

<https://laravel.com/docs/collections#available-methods>

## Crear registros en la base de datos

Para crear un registro en la base de datos se debe crear un nuevo objeto del modelo que se desee crear.

Una vez se disponga de dicho objeto se inicializarán todos los atributos del objeto con los valores que se deseen almacenar en la base de datos.

Por último, se llamará a la función **save()** del modelo de Eloquent que se encargará de insertar los registros en la base de datos.

```

1 $cliente = new Customer();
2 $cliente->nombre = "Guillermo";
3 $cliente->edad = 35;
4 $cliente->email = "g.garridoportes@edu.gva.es";
5 $cliente->save();

```

El controlador de recursos tiene dos funciones encargadas de gestionar la creación de elementos en la base de datos:

**create()** que suele encargarse de proporcionar al usuario una vista que suele ser un formulario vacío para introducir los datos de la tabla.

**store()** que recibe un parámetro de tipo Request o Form Request donde se encuentran todos los elementos que envía el formulario. En este método, si todos los datos son válidos, se creará el objeto utilizando el modelo, se actualizarán los valores con los provenientes en el Request y se insertará en la tabla de la base de datos.

```

1 public function create(){
2     /* Carga el formulario de creación
3      (app/views/customers/create.blade.php) */
4     return View::make('customers.create');
5 }
6
7
8 public function store(Request $request){
9     $cliente = new Customer();
10    $cliente->nombre = $request->nombre;
11    $cliente->edad = $request->edad;
12    $cliente->email = $request->email;
13    $cliente->save();
14 }
```

## Asignación masiva

También en caso que el modelo tenga muchos valores puede ser algo tedioso tener que asignarlos uno a uno. Para facilitar esta tarea se puede utilizar el método **create()** que permite recibir un array con los valores que se deben asignar a cada atributo del objeto.

```

1 public function store(Request $request){
2     Customer::create($request->all());
3 }
```

Para poder realizar esta asignación es imprescindible crear en el modelo, un array llamado **\$fillable** que contendrá aquellos atributos que se pueden llenar de forma masiva así como el orden en que se esperan.

```

1 class Customer extends Model
2 {
3     ...
4     protected $fillable = [
5         'nombre',
6         'edad',
7         'email',
8     ];
```

## Seeders

Otro método de insertar elementos de forma masiva en la base de datos es el uso de los *seeders* o semilleros de información.

Estos son herramientas para inicializar la base de datos con valores predefinidos en la aplicación.

Los *seeders* son muy interesantes para crear la información de la base de datos necesaria para el funcionamiento de la aplicación como podrían ser roles de usuario, categorías, tipos de productos, etcétera. Estos evitan que toda esta información de la base de datos deba ser introducida manualmente o que se necesite exportar e importar las tablas bases de datos constantemente.

Cada *seeder* que se genere será para una determinada tabla de la aplicación, por lo que son archivos que van estrechamente ligados con las migraciones.

Para crear un seeder puede realizarse de forma manual dentro del directorio “**database/seeders**” o bien utilizando el comando de “**php artisan make:seeder**” seguido del nombre del *seeder*.

El nombre del *seeder* por convenio suele ser el nombre de la tabla de la base de datos seguido de “**TableSeeder**”.

```
projeto-ggarrido-laravel> php artisan make:seeder CustomerTableSeeder
INFO Seeder [projeto-ggarrido-laravel\database\Seeder\CustomerTableSeeder.php]
created successfully.
```

Dentro del archivo del seeder generado se puede encontrar el método `run()` que será el que se ejecutará al lanzar los seeders. En este se generarán los diferentes objetos del modelo correspondiente a la tabla a llenar y se insertarán en la base de datos.

Hay que recordar que, para poder crear objetos de un modelo concreto, será necesario importar con la llamada “use” y la ruta con el nombre del modelo que se desea utilizar antes de la creación de la clase.

```
1 <?php
2 namespace Database\Seeders;
3 use Illuminate\Database\Console\Seeds\WithoutModelEvents;
4 use Illuminate\Database\Seeder;
5 use App\Models\Customer;
6 class CustomersTableSeeder extends Seeder{
7     /**
8      * Run the database seeds.
9      * @return void
10     */
11    public function run()    {
12        $cliente = new Customer();
13        $cliente->nombre = "Guillermo";
14        $cliente->edad = 35;
15        $cliente->email = "g.garridoportes@edu.gva.es";
16        $cliente->save();
17
18        $cliente = new Customer();
19        $cliente->nombre = "Silvia";
20        $cliente->edad = 36;
21        $cliente->email = "silvia36@gmail.es";
22        $cliente->save();
23    }
24 }
```

Para que el *seeder* se ejecute además de crear el archivo se deberá añadir, dentro del archivo **DatabaseSeeder.php** que está dentro del mismo directorio que los **seeders**, una llamada al método **call()** que incluirá como primer parámetro un array del nombre de todos los *seeders* que debe cargar.

```
1 public function run()    {
2     $this->call([
3         CustomersTableSeeder::class,
4     ]);
5 }
```

Además, es importante el orden de creación de estos en la base de datos, sobre todo si existen relaciones entre las mismas.

Por último, desde el comando de “**php artisan migrate:fresh**” añadiendo la opción de “**--seed**” creará las tablas desde cero y añadirá los elementos precargados en los *seeders*.

```
projeto-ggarrido-laravel> php artisan migrate:refresh --seed

INFO  Running migrations.

2019_12_14_000001_create_personal_access_tokens_table ..... 26ms DONE
2023_01_11_154311_create_customers_table ..... 6ms DONE
2023_01_11_154554_update_to_customers_table ..... 20ms DONE
2023_01_11_161906_create_addresses_table ..... 6ms DONE
2023_01_11_164156_create_address_lines_table ..... 8ms DONE
2023_01_11_172944_create_social_networks_table ..... 9ms DONE
2023_01_11_173541_create_customer_social_network_table ..... 30ms DONE

INFO  Seeding database.

Database\Seeders\CustomersTableSeeder ..... RUNNING
Database\Seeders\CustomersTableSeeder ..... 267.76 ms DONE
```

| <a href="#">id</a> | <a href="#">nombre</a> | <a href="#">edad</a> | <a href="#">created_at</a> | <a href="#">updated_at</a> | <a href="#">deleted_at</a> | <a href="#">email</a>      |
|--------------------|------------------------|----------------------|----------------------------|----------------------------|----------------------------|----------------------------|
| 1                  | Guillermo              | 35                   | 2023-01-11 18:59:02        | 2023-01-11 18:59:02        | NULL                       | g.garridoportes@edu.gva.es |
| 2                  | Silvia                 | 36                   | 2023-01-11 18:59:02        | 2023-01-11 18:59:02        | NULL                       | silvia36@gmail.es          |

## Model Factories

Otro modo de poder llenar la base de datos es generando objetos usando las fábricas de datos o *factories*.

Los *factories* permiten generar múltiples registros de un modelo generando datos aleatoriamente. Esto es muy útil para poder hacer pruebas de la aplicación y ver con múltiples datos aleatorios como quedaría el diseño de las diferentes vistas y el como se mostrará el contenido en la misma. También pueden servir para realizar test unitarios y pruebas en general de la aplicación, pero no se suelen usar para generar datos en producción.

Para crear un *factory* puede realizarse de forma manual dentro del directorio “database/factories” o bien utilizando el comando de “**php artisan make:factory**” seguido del nombre de la factoría y muy importante pasarle el modelo con la opción “**--model=**” seguido del nombre del modelo que va a utilizar la *factory* para generar las diferentes instancias.

El nombre del *factory* por convenio suele ser el nombre de la tabla de la base de datos seguido de “**Factory**”.

```
projeto-ggarrido-laravel> php artisan make:factory CustomerFactory --model=Customer

INFO  Factory [projeto-ggarrido-laravel\database\factories\CustomerFactory.php]
created successfully.
```

Dentro del *factory* se encontrará el método **definition()** que será el encargado de devolver un array asociativo cuyas claves serán los atributos del modelo con el valor de cada uno de ellos.

Si se pretende llenar la información con datos aleatorios, se puede utilizar un método de la clase *Factory* llamado **faker()** que devolverá un valor del tipo indicado para el elemento del array.

Algunos de los tipos de datos que puede generar **faker()** que genera llamando a sus respectivos métodos son:

- **name();** // Un nombre y apellido de una persona
- **randomDigit();** // Un numero aleatorio
- **word();** // Una palabra
- **sentence();** // Una frase
- **unique()->word();** // Una palabra que no se repita
- **text(\$maxNbChars = 300);** // Un texto de 300 caracteres
- **safeEmail();** // Un email
- **hexcolor();** // Un color hexadecimal
- **randomElement(["elemento1", "elemento2", ...]);** // Un dato aleatorio de los que se encuentren en el array que se le pasa. Muy útil para seleccionar un valor para una clave ajena.

```

1 public function definition()
2 {
3     return [
4         'nombre' => $this->faker->name(),
5         'edad' => $this->faker->randomDigit(),
6         'email' => $this->faker->safeEmail(),
7     ];
8 }
```

Una vez ya está creado el *factory* dentro del archivo **DatabaseSeeder.php** que está dentro del directorio “**database/seeders**”, dentro del método **run()** se podrá llamar al modelo que se desee crear seguido de “**::factory**” y un paréntesis que contendrá el número de elementos. Posteriormente se llamará al método **create()** de *Factory* concatenando con “**->create();**”

```

1 <?php
2 namespace Database\Seeders;
3 use Illuminate\Database\Seeder;
4 use App\Models\Customer;
5 class DatabaseSeeder extends Seeder{
6     /**
7      * Seed the application's database.
8      *
9      * @return void
10     */
11    public function run(){
12        Customer::factory(20)->create();
13    }
14 }

```

Por último, al igual que con los *seeders* se llamará desde el comando de “**php artisan migrate:fresh**” añadiendo la opción de “**--seed**” generará las migraciones, *seeders* y las *factories* que se encuentren en el **DatabaseSeeder.php**.

| <b>id</b> | <b>nombre</b>              | <b>edad</b> | <b>created_at</b>   | <b>updated_at</b>   | <b>deleted_at</b> | <b>email</b>                   |
|-----------|----------------------------|-------------|---------------------|---------------------|-------------------|--------------------------------|
| 1         | Jeramy Blanda              | 2           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | ssauer@example.net             |
| 2         | Mr. Hobart Lindgren        | 4           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | elisha72@example.org           |
| 3         | Marianne Conroy            | 7           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | rachelle.hermiston@example.org |
| 4         | Marilie Monahan            | 7           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | zwelch@example.com             |
| 5         | Soledad Gorczany           | 3           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | oconner.bradly@example.net     |
| 6         | Mr. Sid Senger             | 0           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | era76@example.com              |
| 7         | Sharon Little              | 2           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | vida63@example.net             |
| 8         | Earline DuBuque            | 5           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | jessyca10@example.com          |
| 9         | Anderson West              | 4           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | eileen36@example.com           |
| 10        | Odie Lindgren              | 9           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | cassandra15@example.com        |
| 11        | Julien Boyle               | 5           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | kohler.garret@example.com      |
| 12        | Prof. Viviane Mitchell Sr. | 5           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | kuhic.sydney@example.net       |
| 13        | Sibyl Crooks               | 6           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | kkautzer@example.com           |
| 14        | Esperanza Aufderhar        | 1           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | aryanna58@example.net          |
| 15        | Britney Harber Jr.         | 2           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | qswift@example.com             |
| 16        | Wyman Welch Jr.            | 0           | 2023-01-11 21:56:18 | 2023-01-11 21:56:18 | NULL              | ewuckert@example.org           |
| 17        | Prof. Lester Bechtelar     | 0           | 2023-01-11 21:56:19 | 2023-01-11 21:56:19 | NULL              | kbauch@example.net             |
| 18        | Ferne Ledner               | 8           | 2023-01-11 21:56:19 | 2023-01-11 21:56:19 | NULL              | ward.percival@example.com      |
| 19        | Dr. Marcelino Dickinson MD | 8           | 2023-01-11 21:56:19 | 2023-01-11 21:56:19 | NULL              | ijakubowski@example.com        |
| 20        | Prof. Maybelle Blick II    | 2           | 2023-01-11 21:56:19 | 2023-01-11 21:56:19 | NULL              | hollis.kuhlman@example.com     |



## IMPORTANTE

Para que se pueda usar el método **factory()** de un modelo este debe implementar el *trait* de *Factory* añadiéndolo en la clase “**use HasFactory;**”. Este debe importarse con “**use Illuminate\Database\Eloquent\Factories\HasFactory;**”.

## Leer y listar registros de la base de datos

Para leer un registro de la base de datos se pueden utilizar el generador de consultas (QB) o bien hacer uso de Eloquent y utilizar las colecciones.

De este modo si se desea leer un registro concreto se puede filtrar utilizando la función **find()**, **first()** o mediante condiciones del **where()** y la posterior llamada al método **get()**.

Mientras que si se desea listar todos o parte de los registros se pueden utilizar funciones como **all()**, **chunk()** o **lazy()** entre otros. Incluso se pueden paginar con el método **paginate()** el número de elemento que devolverá la llamada a la base de datos antes de recibir la colección.

Además, si se desean listar los elementos que han sido eliminados utilizando el *softDelete* se podrá hacer uso del método **withTrashed()**.

El controlador de recursos tiene dos funciones encargadas de obtener y mostrar los elementos de la base de datos:

**index()** que suele encargarse de proporcionar al usuario una vista donde se puede visualizar un listado con todos o una parte filtrada de los elementos de la tabla. Para ello realizará una llamada a la base de datos para obtener el listado de registros a mostrar y posteriormente se lo enviará a la vista.

**show()** que recibe un parámetro de tipo integer o del tipo del modelo al que pertenezca. Si recibe un **id** se deberá realizar la llamada ya sea utilizando el método **find()**, **findOrFail()** o cualquier consulta similar.

```
1 public function index(){
2     $clientes = Customer::all();
3     return View::make('customers.index')->with('clientes', $clientes);
4 }
5
6 public function show($id){
7     $cliente = Customer::find($id);
8     return View::make('customers.show')->with('cliente', $cliente);
9 }
```

Si se indica el tipo del modelo que se va a recibir en la función (y se le pasa un **id** desde la vista) Laravel ejecutará automáticamente la llamada a la base de datos obteniendo dicho elemento utilizando el método **findOrFail()**. En caso que no encuentre un registro con dicho **id** redirigirá al usuario a una página de error 404.

```
1 public function show(Customer $cliente){
2     return View::make('customers.show')->with('cliente', $cliente);
3 }
```

## Actualizar y eliminar registros de la base de datos

La actualización de registros de la base de datos es muy sencilla ya que prácticamente funciona igual que la creación de estos. La diferencia principal entre una inserción y una actualización es que en el segundo caso el objeto no se debe generar vacío, sino que se lee desde la base de datos, por lo que dicho objeto contara con el atributo **id** con un valor específico.

Una vez obtenido el objeto, se modifican los valores deseados (menos el **id**) de los atributos del objeto y se llama a la función **save()** para que modifique el registro.

El controlador de recursos tiene dos funciones encargadas de actualizar los elementos de la base de datos:

**edit()** que suele encargarse de proporcionar al usuario una vista al que se le pasa el registro a editar y que suele cargar en un formulario los valores del mismo para modificar los datos de este.

**update()** que recibe un parámetro de tipo *Request* o *Form Request* donde se encuentran todos los elementos que envía el formulario. Además, recibe un id del registro a modificar. De este modo la función buscara el registro en la base de datos, modificará los valores con los provenientes del formulario y llamará a la función **save()** que almacenará los cambios.

```

1 public function edit($id){
2     $cliente = Customer::find($id);
3     return View::make('customers.edit')->with('cliente', $cliente);
4 }
5
6 public function update(Request $request, $id){
7     $cliente = Customer::find($id);
8     $cliente->nombre = $request->nombre;
9     $cliente->edad = $request->edad;
10    $cliente->email = $request->email;
11    $cliente->save();
12 }
```

Por otro lado, para eliminar un registro, se lee el registro a eliminar de la base de datos y posteriormente se llama a la función **delete()** que se encargará de eliminarlos.

Si se está haciendo uso del *SoftDelete* también será posible volver a restaurar un registro eliminado haciendo uso del método **restore()**. Pero si se desea eliminar completamente la información de la tabla aunque se esté usando el *SoftDelete*, se podrá hacer uso de la función **forceDelete()**;

El controlador de recursos cuenta con la función **destroy()** que recibe un **id** y se encarga de borrar el elemento que tenga el identificador en la tabla de la base de datos.

```

1 public function destroy($id){
2     $cliente = Customer::find($id);
3     $cliente->delete();
4 }
```

## 4. Formularios

Un formulario se define empleando HTML junto con opciones de blade.

Por convención todas las vistas que interactúan o se cargan desde el controlador de recursos de dicho modelo, suelen estar agrupadas en una carpeta con el nombre del modelo en plural

En el action del formulario se deberá llamar a la ruta correspondiente para procesar los datos del mismo.

Laravel cuenta con la directiva **@csrf** (*cross-site request forgeries*), que añade un campo oculto con un token de validación del usuario protegiendo de ataques **XSS** (*cross-site scripting*) de suplantación de identidad, y obteniendo un error de tipo 419 si se envía un formulario no validado.

Por otra parte, en el atributo **action** se debe establecer la ruta donde se recibirán los datos enviados por el formulario en el controlador, el cual recibirá un parámetro de tipo *Request* donde se almacenarán todos los datos del formulario. Normalmente el controlador de recursos cuenta con dos métodos que reciben un parámetro de tipo *Request*, **store()** y **update()**.

```

1 <form method="post" action="{{ route('customers.store') }}>
2   @csrf
3   <div>
4     <label>Nombre</label>
5     <input type="text" name="nombre" id="nombre">
6   </div>
7   <div>
8     <label>Edad</label>
9     <input type="number" name="edad" id="edad">
10  </div>
11  <div>
12    <label>Email</label>
13    <input type="email" name="email" id="email">
14  </div>
15  <input type="submit" name="send" value="Enviar">
16 </form>
```

## Validar formularios

Las validaciones en las aplicaciones web siempre deben realizarse tanto en el lado del cliente, como en el lado del servidor. De este modo, se previene que los datos puedan provenir de una fuente diferente a un formulario por lo que podrían no haber sido validados.

Para la realización de esta validación, se puede hacer uso del método **validate()** que dispone la clase Request. Para ello se generará un array asociativo de reglas, con los campos a validar y los requisitos de los mismos. Este servirá como argumento en la función de **validate()**.

Para poder concatenar más de una regla para un mismo campo se utilizará el símbolo tubería (|) que se encargará de unir las diferentes reglas. De este modo la primera regla de un campo que falle, devolverá error y no continuara la validación del resto.



## ENLACE

En el enlace “Reglas de validación” se pueden consultar una lista de las posibles validaciones que se pueden utilizar para validar los campos de los formularios desde el controlador. <https://laravel.com/docs/validation#available-validation-rules>

```

1 $reglas = [
2   'nombre' => 'required|min:2',
3   'edad' => 'required',
4   'email' => 'required|email'
5 ];
6 $request->validate($reglas);

```

Como segundo parámetro, la función **validate()** también acepta un array de mensajes o textos de error que se mostrarán en caso que alguno de los campos no sea válido.

Si la validación falla, este método lanzará una excepción de tipo **Illuminate\Validation\ValidationException**, de modo que Laravel redireccionará al usuario a la ruta donde se encontraba anteriormente, normalmente el formulario, con los mensajes de error producidos.

```

1 $reglas = [
2   'nombre' => 'required|min:2',
3   'edad' => 'required',
4   'email' => 'required|email'
5 ];
6 $mensajes_de_error = [
7   'nombre' => 'El nombre debe contener al menos 2 caracteres.',
8   'edad' => 'La edad es obligatoria.',
9   'email' => 'El email es necesario y debe ser válido.'
10 ];
11 $request->validate($reglas, $mensajes_de_error);

```

La validación puede hacerse directamente desde el método que recibe los datos del formulario. Pero para desacoplar el código adecuadamente existe una alternativa llamada **Form Request** que son clases que se encargan de agrupar la lógica de validación de los formularios. Estas son muy útiles para poder reutilizar el mismo formulario en distintas llamadas, ya que omiten el tener código redundante en los controladores.

Para generar un *Form Request* se puede añadir de forma manual dentro del directorio “`app/Http/Request`” o bien utilizando el comando de “`php artisan make:request`” y el nombre del modelo al que validará (en caso de tener validaciones distintas para el mismo modelo se podrá indicar el nombre del método concatenándolo posteriormente) y por último la palabra *Request*.

```
proyecto-ggarrido-laravel> php artisan make:request CustomerRequest
INFO Request [proyecto-ggarrido-laravel_mysql\app\Http\Requests\CustomerRequest.php]
created successfully.
```

El archivo que se genera contiene tres métodos:

- **`authorize()`**: que devolverá un booleano indicando si se requiere autorización para utilizar esta *Request* (`false`), o no (`true`).
- **`rules()`**: donde se indicarán las reglas de validación.
- **`messages()`**: donde se podrán personalizar los mensajes de error.

```
1 <?php
2 namespace App\Http\Requests;
3 use Illuminate\Foundation\Http\FormRequest;
4 class CustomerRequest extends FormRequest
5 {
6     public function authorize()
7     {
8         return true;
9     }
10    public function rules()
11    {
12        return [
13            'nombre' => 'required|min:2',
14            'edad' => 'required',
15            'email' => 'required|email'
16        ];
17    }
18    public function messages()
19    {
20        return [
21            'nombre' => 'El nombre debe contener al menos 2 caracteres.',
22            'edad' => 'La edad es obligatoria.',
23            'email' => 'El email es necesario y debe ser válido.'
24        ];
25    }
26 }
```

Para poder utilizar un *Form Request* simplemente hay que indicarlo en el método del controlador, cuando se pasa el tipo de dato para el parámetro de entrada. De este modo la validación se produce automáticamente antes de llegar a ejecutar el código del método.

```

1 public function store(CustomerRequest $request){
2     ...
3 }

```

## Mostrar errores de validación de formularios

Si después de llamar al método de **validate()**, se produce un error de validación, esta devolverá una instancia de **Illuminate\Support\MessageBag**, que quedará almacenada en la variable **\$errors**. Esta variable está disponible automáticamente desde cualquier lugar de Laravel.

La clase *MessageBag* tiene diferentes métodos que permiten comprobar y mostrar los errores recibidos, como el método **any()**, que comprueba si hay algún error y el método, **all()** que devuelve un array con los errores producidos.

```

1 @if ($errors->any())
2     <ul>
3         @foreach($errors->all() as $error)
4             <li>{{ $error }} </li>
5         @endforeach
6     </ul>
7 @endif

```

Del mismo modo, si en lugar de mostrar todo el array de errores se pretende mostrar un error que pertenece a un campo específico, se puede hacer uso del método **has()** que comprobara si un campo concreto tiene un error y **first()** para obtener el mensaje de error del campo. Ambos métodos recibirán como primer parámetro el nombre del campo.

```

1 <div>
2     <label>Nombre</label>
3     <input type="text" name="nombre" id="nombre">
4     @if ($errors->has('nombre'))
5         <div class="error">
6             {{ $errors->first('nombre') }}
7         </div>
8     @endif
9 </div>

```

## Cargar y recargar información en los formularios.

Además de mostrar los errores, si queremos que los datos enviados se carguen nuevamente en los inputs, mejorando la experiencia del usuario de modo que pueda ver que introdujo, se puede hacer uso del método **old()** de Laravel que también recibirá como primer parámetro el nombre del input permitiendo mostrar el valor antiguo en el atributo **value**.

```

1 <div>
2   <label>Nombre</label>
3   <input type="text" name="nombre" id="nombre" value="{{ old('nombre') }}>
4   @if ($errors->has('nombre'))
5     <div class="error">
6       {{ $errors->first('nombre') }}
7     </div>
8   @endif
9 </div>
```

Si lo que se desea es cargar la información proveniente de una variable, por ejemplo un objeto de algún modelo, que se le pase a la vista en lugar de cargar información introducida anteriormente, como suele ocurrir en formularios de edición por ejemplo, se puede hacer cargando el atributo del modelo que se le pase a la vista en el atributo *value*.

```

1 <div>
2   <label>Nombre</label>
3   <input type="text" name="nombre" id="nombre" value="{{ $customer->nombre }}>
4   @if ($errors->has('nombre'))
5     <div class="error">
6       {{ $errors->first('nombre') }}
7     </div>
8   @endif
9 </div>
```

Además, es posible que en las vistas existan enlaces a rutas del sistema que reciben como parámetro un identificador. Laravel permite pasarle el objeto entero en la ruta del enlace y este enviará únicamente el campo **id** del objeto cuando se ejecute la llamada.

```

1 <form method="post" action="{{ route('customers.update', $customer) }}>
2   @csrf
3   ...
4 </form>
```