CLASE 3 / 6

Indice

Base de Datos

Laravel facilita la configuración y el uso de diferentes tipos de base de datos: MySQL, PostgreSQL, SQLite y SQL Server. En el archivo de configuración (`config/database.php`) se indican todos los parámetros de acceso a la bases de datos y además se especifica cual es la conexión que se utilizará por defecto. En Laravel se pueden hacer uso de varias bases de datos a la vez, aunque sean de distinto tipo. Por defecto se accederá a la que se especifique en la configuración y si se desea acceder a otra conexión se deberá indicar expresamente al realizar la consulta.

Configuración de la Base de Datos

Lo primero es completar la configuración de la Base de Datos. Como ejemplo se va a configurar el acceso a una base de datos tipo MySQL. Al editar el archivo con la configuración `config/database.php` se ve en primer lugar la siguiente línea:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

Este valor indica el tipo de base de datos a utilizar por defecto. Laravel utiliza el sistema de variables de entorno para separar las distintas configuraciones de usuario o de máquina. El método `env('DB_CONNECTION', 'mysql')` lo que hace es obtener el valor de la variable `DB_CONNECTION` del archivo `.env`. En caso de que dicha variable no esté definida devolverá el valor por defecto `mysql`.

En este mismo archivo de configuración, dentro de la sección `connections`, se encuentran todos los campos utilizados para configurar cada tipo de base de datos, en concreto la base de datos tipo `mysql` tiene los siguientes valores:

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'database'  => env('DB_DATABASE', 'forge'), // Nombre de la bd
    'username'  => env('DB_USERNAME', 'forge'), // Usuario de acceso a bd
    'password'  => env('DB_PASSWORD', ''),      // Contraseña de acceso
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
    'strict'    => false,
],
```

Como se puede ver, básicamente los campos a configurar para usar la base de datos son: *host*, *database*, *username* y *password*. El *host* se dejará como está si se usará una base de datos local, mientras que los otros tres campos sí que hay que actualizarlos con el nombres de la base de datos a utilizar y el usuario y la contraseña de acceso. Para poner estos valores se hace desde el archivo `.env` de la raíz del proyecto:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_DATABASE=nombre-base-de-datos
DB_USERNAME=nombre-de-usuario
DB_PASSWORD=contraseña-de-acceso
```

CREAR LA BASE DE DATOS

Para crear la base de datos en MySQL utilizar la herramienta *PHPMyAdmin* que se ha instalado con el paquete XAMPP. Para esto acceder por la ruta:

```
http://localhost/phpmyadmin
```

Mostrará un panel para la gestión de las bases de datos de MySQL, que permite, además de realizar cualquier tipo de consulta SQL, crear nuevas bases de datos o tablas, e insertar, modificar o eliminar los datos directamente. En este caso se ingresara en la pestaña "Bases de datos" y creará una nueva base de datos. El nombre tiene que ser el mismo indicado en el archivo de configuración de Laravel.

TABLA DE MIGRACIONES

A continuación se debe crear la tabla de migraciones. Laravel utiliza las migraciones para poder definir y crear las tablas de la base de datos desde código, y de esta manera tener un control de las versiones de las mismas.

Para poder empezar a trabajar con las migraciones es necesario en primer lugar crear la tabla de migraciones. Para esto ejecutar el siguiente comando de Artisan:

```
php artisan migrate:install
```

> Si diese algún error, revisar la configuración que se ha puesto de la base de datos y si se ha creado la base de datos con el nombre, usuario y contraseña indicado. También puede dar error si no esta instalado el paquete mysql para la versión php `sudo apt install php-mysql`

Si todo funciona correctamente se podrá ir al navegador y acceder de nuevo a la base de datos con PHPMyAdmin, ver que se habrá creado la tabla `migrations`. Con esto ya esta configurada la base de datos y el acceso a la misma.

MIGRACIONES

Las migraciones son un sistema de control de versiones para bases de datos. Permiten que un equipo trabaje sobre una base de datos añadiendo y modificando campos, manteniendo un histórico de los cambios realizados y del estado actual de la base de datos. Las migraciones se utilizan de forma conjunta con la herramienta *Schema builder* para gestionar el esquema de base de datos de la aplicación.

La forma de funcionar de las migraciones es crear archivos (PHP) con la descripción de la tabla a crear y posteriormente, si se quiere modificar dicha tabla se añadiría una nueva migración (un nuevo archivo PHP) con los campos a modificar. Artisan incluye comandos para crear migraciones, para ejecutar las migraciones o para hacer *rollback* de las mismas (volver atrás).

Crear una nueva migración

Para crear una nueva migración se utiliza el comando de Artisan `make:migration`, al cual se le pasará el nombre del archivo a crear y el nombre de la tabla:

```
php artisan make:migration create_alumnos_table --create=alumnos
```

Esto nos creará un archivo de migración en la carpeta `database/migrations` con el nombre `<TIMESTAMP>_create_alumnos_table.php`. Al añadir un *timestamp* a las migraciones el sistema sabe el orden en el que tiene que ejecutar (o deshacer) las mismas.

Si lo que queremos es añadir una migración que modifique los campos de una tabla existente tendremos que ejecutar el siguiente comando:

```
php artisan make:migration add_to_alumnos_table --table=alumnos
```

En este caso se creará también un archivo en la misma carpeta, con el nombre `<TIMESTAMP>_add_to_alumnos_table.php` pero preparado para modificar los campos de dicha tabla.

Por defecto, al indicar el nombre del archivo de migraciones se suele seguir siempre el mismo patrón (aunque en realidad el nombre es libre). Si es una migración que crea una tabla el nombre tendrá que ser `create_<table-name>_table` y si es una migración que modifica una tabla será `<action>_to_<table-name>_table`.

Estructura de una migración

El archivo o clase PHP generada para una migración siempre tiene una estructura similar a la siguiente:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateAlumnosTable extends Migration
{
    /**
     * Run the migrations.
     * @return void
     */
    public function up()
    {
        //
    }

    /**
     * Reverse the migrations.
     * @return void
     */
    public function down()
    {
        //
```

```
    }
}
```

En el método `up` es donde se tendrá que crear o modificar la tabla, y en el método `down` se deberá de deshacer los cambios que se hagan en el `up` (eliminar la tabla o eliminar el campo que se haya añadido). Esto permitirá poder ir añadiendo y eliminando cambios sobre la base de datos y tener un control o histórico de los mismos.

EJECUTAR MIGRACIONES

Después de crear una migración y de definir los campos de la tabla se tendrá que lanzar la migración con el siguiente comando:

```
php artisan migrate
```

Si aparece el error "`class not found`" se podrá solucionar llamando a `composer dump-autoload` y volviendo a lanzar las migraciones.

Este comando aplicará la migración sobre la base de datos. Si hubiera más de una migración pendiente se ejecutarán todas. Para cada migración se llamará a su método `up` para que cree o modifique la base de datos. Posteriormente en caso de querer deshacer los últimos cambios se puede ejecutar:

```
php artisan migrate:rollback

# O si se desea deshacer todas las migraciones
php artisan migrate:reset
```

Un comando interesante cuando se esta desarrollando un nuevo sitio web es `migrate:refresh`, el cual deshará todos los cambios y volverá a aplicar las migraciones o aplicara los cambios en la estructura de las tablas que se hayan realizado por codigo (ojo destruye todos los registros ya cargados):

```
php artisan migrate:refresh
```

Además para comprobar el estado de las migraciones, para ver las que ya están instaladas y las que quedan pendientes, se ejecuta:

```
php artisan migrate:status
```

SCHEMA BUILDER

Una vez creada una migración se deberá completar sus métodos `up` y `down` para indicar la tabla que se desea crear o el campo que se quiere modificar. En el método `down` siempre se añadirá la operación inversa, eliminar la tabla que se ha creado en el método `up` o eliminar la columna que se ha añadido. Esto permitirá deshacer migraciones dejando la base de datos en el mismo estado en el que se encontraban antes de que se añadieran.

Para especificar la tabla a crear o modificar, así como las columnas y tipos de datos de las mismas, se utiliza la clase *Schema*. Esta clase tiene una serie de métodos que permitirá especificar la estructura de las tablas independientemente del sistema de base de datos que se utilice.

CREAR Y BORRAR UNA TABLA

Para añadir una nueva tabla a la base de datos se utiliza el siguiente constructor:

```
Schema::create('alumnos', function (Blueprint $table) {
    $table->increments('id');
});
```

Donde el primer argumento es el nombre de la tabla y el segundo es una función que recibe como parámetro un objeto del tipo *Blueprint* que se utiliza para configurar las columnas de la tabla.

En la sección `down` de la migración se eliminará la tabla que se ha creado con alguno de los siguientes métodos:

```
Schema::drop('alumnos');

Schema::dropIfExists('alumnos');
```

Al crear una migración con el comando de Artisan `make:migration` ya viene este código añadido por defecto, la creación y eliminación de la tabla que se ha indicado y además se añaden un par de columnas por defecto (*id* y *timestamps*).

AÑADIR COLUMNAS

El constructor `Schema::create` recibe como segundo parámetro una función que permite especificar las columnas que va a tener dicha tabla. En esta función se añaden todos los campos que se requieren, indicando para cada uno de ellos su tipo y nombre, y además también se podrá indicar una serie de modificadores como valor por defecto, índices, etc. Por ejemplo:

```
Schema::create('alumnos', function($table)
{
    $table->increments('id');
    $table->string('nombre_apellido', 32);
    $table->string('curso');
    $table->smallInteger('anyo');
    $table->string('email');
    $table->boolean('pago_inscripcion')->default(false);
    $table->timestamps();
});
```

Schema define muchos tipos de datos que pueden utilizarse para definir las columnas de una tabla, algunos de los principales son:

Comando	Tipo de campo
`$table->boolean('confirmed');`	BOOLEAN
`$table->enum('choices', array('foo', 'bar'));`	ENUM
`$table->float('amount');`	FLOAT

`$table->increments('id');`	Clave principal INTEGER con autoincremental
`$table->integer('votes');`	INTEGER
`$table->mediumInteger('numbers');`	MEDIUMINT
`$table->smallInteger('votes');`	SMALLINT
`$table->tinyInteger('numbers');`	TINYINT
`$table->string('email');`	VARCHAR
`$table->string('name', 100);`	VARCHAR con longitud limitada
`$table->text('description');`	TEXT
`$table->timestamp('added_on');`	TIMESTAMP
`$table->timestamps();`	Añade los timestamps "created_at" y "updated_at"
`->nullable()`	Indicar que la columna permite valores NULL
`->default($value)`	Declara el valor por Default de la columna
`->unsigned()`	Añade UNSIGNED a las columnas INTEGER

Los tres últimos se pueden combinar con el resto de tipos para crear, por ejemplo, una columna que permita nulos, con un valor por defecto y de tipo *unsigned*.

Para eliminar las tablas agregadas en el metodo down

```
$table->dropColumn('nombre_de_tabla');
```

Para consultar todos los tipos de datos que se pueden utilizar, consultar la documentación de Laravel en: https://laravel.com/docs/6.x/migrations#creating-columns

AÑADIR ÍNDICES

Schema soporta los siguientes tipos de índices:

Comando	Descripción
`$table->primary('id');`	Añadir una clave primaria
`$table->primary(array('first', 'last'));`	Definir una clave primaria compuesta
`$table->unique('email');`	Definir el campo como UNIQUE
`$table->index('state');`	Añadir un índice a una columna

En la tabla se especifica como añadir estos índices después de crear el campo, pero también permite indicar estos índices a la vez que se crea el campo:

```
$table->string('email')->unique();
```

CLAVES FORANEAS

Con *Schema* también pueden definirse claves foráneas entre tablas:

```
$table->integer('user_id')->unsigned();
$table->foreign('user_id')->references('id')->on('users');
```

En este ejemplo en primer lugar se añade la columna "`user_id`" de tipo UNSIGNED INTEGER (siempre se debe crear primero la columna sobre la que se va a aplicar la clave foranea). A continuación se crea la clave foránea entre la columna "`user_id`" y la columna "`id`" de la tabla "`users`".

La columna con la clave foránea tiene que ser **del mismo tipo** que la columna a la que apunta. Si por ejemplo se crea una columna con índice auto-incremental se deberá especificar que la columna sea *unsigned* para que no se produzcan errores.

También se puede especificar las acciones que se tienen que realizar para "*on delete*" y "*on update*":

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

Para eliminar una clave foránea, en el método `down` de la migración se utilizará el siguiente código:

```
$table->dropForeign('posts_user_id_foreign');
```

Para indicar la clave foránea a eliminar se emplea el siguiente patrón para especificar el nombre `<tabla>_<columna>_foreign`. Donde "tabla" es el nombre de la tabla actual y "columna" el nombre de la columna sobre la que se creo la clave foránea.

MODIFICAR UNA TABLA

Para poder realizar una modificacion en la estructura de una tabla se deben crear migraciones adicionales.

```
php artisan make:migration add_to_alumnos_table --table=alumnos
```

Lo que generara automaticamente una migracion cuyo schema sera

```
public function up()
    {
        Schema::table('alumnos', function (Blueprint $table) {
            //
        });
    }
```

Al ejecutar esta migracion, no eliminara los registros y agregara el campo especificado.

```
php artisan migrate
```

INICIALIZACIÓN DE LA BASE DE DATOS (*DATABASE SEEDING*)

Laravel también facilita la inserción de datos iniciales o datos *semilla* en la base de datos. Esta opción es muy útil para tener datos de prueba cuando se esta desarrollando una web o para crear tablas que ya tienen que contener una serie de datos en producción.

Los archivos de "semillas" se encuentran en la carpeta `database/seeds`. Por defecto Laravel incluye el archivo `DatabaseSeeder` con el siguiente contenido:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     * @return void
     */
    public function run()
    {
        //...
    }
}
```

Al lanzar la incialización se llamará por defecto al método `run` de la clase `DatabaseSeeder`. Desde aquí se pueden crear las semillas de varias formas:

1. Escribir el código para insertar los datos dentro del propio método `run`.
2. Crear otros métodos dentro de la clase `DatabaseSeeder` y llamarlos desde el método `run`. De esta forma se separan mejor las inicializaciones.
3. Crear otros archivos *Seeder* y llamarlos desde el método `run` de la clase principal.

A continuación se incluye un ejemplo de la opción 1:

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        // Borramos los datos de la tabla
        DB::table('docentes')->delete();

        // Añadimos una entrada a esta tabla
        docentes::create(array('email' => 'prof@instituto.com'));
    }
}
```

Como se puede ver en el ejemplo en general se tendrá que eliminar primero los datos de la tabla en cuestión y posteriormente añadir los datos. Para insertar datos en una tabla se puede utilizar el método que se usa en el ejemplo o alguna de las otras opciones que se verán en las siguientes secciones sobre "Constructor de consultas" y "Eloquent ORM".

CREAR ARCHIVOS SEMILLA

Se pueden crear más archivos o clases *semilla* para modularizar mejor el código de las inicializaciones. De esta forma se puede crear un archivo de semillas para cada una de las tablas o modelos de datos que haya en la aplicación.

En la carpeta `database/seeds` se pueden añadir más archivos PHP con clases que extiendan de `Seeder` para definir propios archivos de "semillas". El nombre de los archivos suele seguir el mismo patrón `<nombre-tabla>TableSeeder`, por ejemplo "`MateriasTableSeeder`". Artisan incluye un comando que facilita crear los archivos de semillas y que además incluirán las estructura base de la clase. Por ejemplo, para crear el archivo de inicialización de la tabla de alumnos:

```
php artisan make:seeder AlumnosTableSeeder
```

Para que esta nueva clase se ejecute hay que llamarla desde el método `run` de la clase principal `DatabaseSeeder` de la forma:

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        Model::unguard();

        $this->call(AlumnosTableSeeder::class);

        Model::reguard();
    }
}
```

El método `call` lo que hace es llamar al método `run` de la clase indicada. Además en el ejemplo se ha añadido las llamadas a `unguard` y a `reguard`, que lo que hacen es desactivar y volver a activar (respectivamente) la inserción de datos masiva o por lotes.

EJECUTAR LA INICIALIZACIÓN DE DATOS

Una vez definidos los archivos de semillas, cuando se desee ejecutarlos para rellenar de datos la base de datos se debera usar el siguiente comando de Artisan:

```
php artisan db:seed
```

MODELOS DE DATOS MEDIANTE ORM

El mapeado objeto-relacional (más conocido por su nombre en inglés, *Object-Relational mapping*, o por sus siglas ORM) es una técnica de programación para convertir datos entre un lenguaje de programación orientado a objetos y una base de datos relacional como motor de persistencia. Esto posibilita el uso de las características propias de la orientación a objetos, posibilitando acceder directamente a los campos de un objeto para leer los datos de una base de datos o para insertarlos o modificarlos.

Laravel incluye su propio sistema de ORM llamado *Eloquent*, el cual proporciona una manera elegante y fácil de interactuar con la base de datos. Para cada tabla de la base datos se tendrá que definir su correspondiente modelo, el cual se utilizará para interactuar desde el código con la tabla.

La forma de trabajo de Eloquent implementa el patrón "Active Record", un patrón de arquitectura de software que permite almacenar en bases de datos relacionales el contenido de los objetos que se tiene en memoria. Esto se hace por medio de métodos como save(), update() o delete(), provocando internamente la escritura en la base de datos, pero sin que se tenga que especificar las propias sentencias.

En Active Record una tabla está directamente relacionada con una clase. En ese caso se dice que la clase es una envoltura de la tabla. La clase en si es lo que conocemos en el framework como "modelo". Cuando creamos un nuevo objeto de ese modelo y se decide salvarlo, se produce la creación de un registro de la tabla. Cuando el objeto se modifica y se salvan los datos, se produce el correspondiente update en ese registro. Cuando ese objeto se borra, se produce el delete sobre ese registro de la tabla.

ORM sería entonces la herramienta de persistencia y Active Record el patrón de arquitectura que se sigue para su construcción. Eloquent es el nombre con el que se conoce en Laravel esta parte del framework, que agiliza la mayoría de las operaciones habituales del acceso a bases de datos.

DEFINICIÓN DE UN MODELO

Por defecto los modelos se guardarán como clases PHP dentro de la carpeta `app`, sin embargo Laravel da libertad para colocarlos en otra carpeta si se requiere, como por ejemplo la carpeta `app/Models`. Pero en este caso hay que asegurarse de indicar correctamente el espacio de nombres.

Para definir un modelo que use *Eloquent* únicamente se deberá crear una clase que herede de la clase `Model`:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    //...
}
```

Sin embargo es mucho más fácil y rápido crear los modelos usando el comando `make:model` de Artisan:

```
php artisan make:model User
```

Este comando creará el fichero `User.php` dentro de la carpeta `app` con el código básico de un modelo. A pesar que está prácticamente vacío de código es importante señalar que los modelos de por si ya hacen bastante cosas, gracias a que heredan el comportamiento a partir de la clase `Illuminate\Database\Eloquent\Model`.

Tambien es posible crear el modelo y la migracion para crear la estructura de la tabla a la que hara referencia utilizando

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

CONVENIOS EN *ELOQUENT*

Nombre

En general el nombre de los modelos se pone en singular con la primera letra en mayúscula, mientras que el nombre de las tablas suele estar en plural y minusculas. Gracias a esto, al definir un modelo no es necesario indicar el nombre de la tabla asociada, sino que *Eloquent* automáticamente buscará la tabla transformando el nombre del modelo a minúsculas y buscando su plural (en inglés).

Si la tabla tuviese otro nombre se podrá indicar usando la propiedad protegida `$table` del modelo:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $table = 'my_users';
}
```

Clave primaria

Laravel también asume que cada tabla tiene declarada una clave primaria autoincremental con el nombre `id`. En el caso de que no sea así, se tendrá que sobrescribir el valor de la propiedad protegida `$primaryKey` del modelo, por ejemplo: `protected $primaryKey = 'my_id';`.

> Es importante definir correctamente este valor ya que se utiliza en determinados métodos de *Eloquent*, como por ejemplo para buscar registros o para crear las relaciones entre modelos.

Timestamps

Otra propiedad que en ocasiones se tienen que establecer son los *timestamps* automáticos. Por defecto *Eloquent* asume que todas las tablas contienen los campos `updated_at` y `created_at` (los cuales se añaden muy fácilmente con *Schema* añadiendo `$table->timestamps()` en la migración). Estos campos se actualizarán automáticamente cuando se cree un nuevo registro o se modifique. En el caso de que no sean utilizados (y que no estén añadidos a la tabla) se deberá indicarlo en el modelo o de otra forma daría un error. Para indicar que no los actualice automáticamente se tendrá que modificar el valor de la propiedad pública `$timestamps` a *false*, por ejemplo: `public $timestamps = false;`.

A continuación se muestra un ejemplo de un modelo de *Eloquent* en el que se añaden todas las especificaciones vistas:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $table = 'my_users';
    protected $primaryKey = 'my_id'
    public $timestamps = false;
}
```

USO DE UN MODELO DE DATOS

Una vez creado el modelo ya se puede utilizar para recuperar datos de la base de datos, para insertar nuevos datos o para actualizarlos. El sitio correcto donde realizar estas acciones es en el controlador, el cual se los tendrá que pasar a la vista ya preparados para su visualización.

Es importante que para su utilización se indique al inicio de la clase el espacio de nombres del modelo o modelos a utilizar. Por ejemplo, si se va a usar los modelos `User` y `Orders` se tendrá que añadir:

```
use App\User;
use App\Orders;
```

CONSULTAR DATOS

Para obtener todas las filas de la tabla asociada a un modelo se usará el método `all()`:

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

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

Este método devolverá un array de resultados, donde cada item del array será una instancia del modelo `User`. Gracias a esto al obtener un elemento del array se puede acceder a los campos o columnas de la tabla como si fueran propiedades del objeto (`$user->name`).

> Nota: Todos los métodos que se describen en la sección de "Constructor de consultas" y en la documentación de Laravel sobre "Query Builder" también se pueden utilizar en los modelos Eloquent. Por lo tanto se podran utilizar *where, orWhere, first, get, orderBy, groupBy, having, skip, take,* etc. para elaborar las consultas.

Eloquent también incorpora el método `find($id)` para buscar un elemento a partir del identificador único del modelo, por ejemplo:

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

En concreto lo que devuelve esta función es un modelo, un objeto de la clase App\User. Como es un modelo dispondremos de los métodos que nos facilitan los modelos de Eloquent, pero también se puede usar como si fuera un simple array con los valores del elemento obtenido.

Si queremos que se lance una excepción cuando no se encuentre un modelo utilizar los métodos `findOrFail` o `firstOrFail`. Esto permite capturar las excepciones y mostrar un error 404 cuando sucedan.

```
$model = User::findOrFail(1);

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

A continuación se incluyen otros ejemplos de consultas usando Eloquent:

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

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

//Obtener 10 primeros artículos, ordenados por fecha, cuyo nombre incluyan
introduccion
$articulos = Articulo::where('nombre_articulo', 'like', 'introduccion%')
            ->orderBy('fecha', 'desc')
            ->take(10)
            ->get();
```

También se pueden utilizar los métodos agregados para calcular el total de registros obtenidos, o el máximo, mínimo, media o suma de una determinada columna. Por ejemplo:

```
$count = User::where('votes', '>', 100)->count();
$price = Orders::max('price');
$price = Orders::min('price');
$price = Orders::avg('price');
$total = User::sum('votes');
```

INSERTAR DATOS

Para añadir una entrada en la tabla de la base de datos asociada con un modelo simplemente se tendrá que crear una nueva instancia de dicho modelo, asignar los valores deseados y por último guardarlos con el método `save()`:

```
$user = new User;
$user->name = 'Matias';
$user->email = 'matias@gmail.com';
$user->save();
```

Para obtener el identificador asignado en la base de datos después de guardar (cuando se t-ate de tablas con índice auto-incremental), recuperar simplemente accediendo al campo `id` del objeto que se habia creado, por ejemplo:

```
$insertedId = $user->id;
```

ACTUALIZAR DATOS

Para actualizar una instancia de un modelo se tendrá que recuperar en primer lugar la instancia que se desea actualizar, a continuación modificarla y por último guardar los datos:

```
$user = User::find(1);
$user->email = 'matiasgarcia@gmail.com';
$user->save();
```

BORRAR DATOS

Para borrar una instancia de un modelo en la base de datos simplemente se tendrá que usar su método `delete()`:

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

Si por ejemplo se desea borrar un conjunto de resultados también se puede usar el método `delete()`de la forma:

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

RELACIONES SOPORTADAS POR ELOQUENT

Existen diversos tipos de relaciones en bases de datos, las básicas que todo el mundo debe conocer son:

- Relaciones de uno a uno: Por ejemplo, un usuario tiene un perfil de usuario. El usuario tiene un perfil y el perfil solo pertenece a un usuario.
- De uno a muchos (1 a n): Por ejemplo, un usuario tiene posts. El usuario puede cargar un número indeterminado de post, pero un post sólo pertenece a un usuario.
- De muchos a muchos (n a m): Por ejemplo, un artículo tiene varias etiquetas y una etiqueta puede tener varios artículos. Estas relaciones generan una tabla adicional generalmente que Eloquent maneja para ti.

Eloquent soporta otras relaciones no tan comunes pero que también se encuentran en la vida real.

- Relaciones de pertenencia a través de otras entidades: En un esquema relacional la tabla "a" se relaciona con "b" y a su vez "b" se relaciona con "c". En un caso como este, con Eloquent se puede conseguir definir una relación directa desde "a" hacia "c", pasando a través de "b". Ese paso "a través" se realiza de manera transparente para el desarrollador. Es decir, es como si la tabla "a" estuviera directamente vinculada a "c".
- Relaciones polimórficas: Básicamente consiste en una relación donde aquella entidad con la que se esta relacionando pueda ser variable. Por ejemplo, un usuario puede dar el "me gusta" a post de otro usuario, a un comentario de otro usuario, a un vídeo, a un artículo en venta… en general, a un número de entidades indeterminado y variable. De ahí el polimorfismo.

Hay que recordar que las relaciones en la base de datos se definen a la hora de hacer migraciones. Dependiendo del tipo de relación esta declaración de función se realizará de una manera o de otra.

DECLARACIÓN DE LAS RELACIONES PARA LA CONFIGURACIÓN DE LOS MODELOS

En Laravel, la declaración de relaciones se realiza mediante una función, que se suele colocar en ambos modelos pertenecientes a ambas tablas relacionadas. No obstante, no es una necesidad realizar estas funciones, puesto que solo harán falta si realmente se necesita acceder desde un modelo a su información relacionada.

Todas las funciones responden a un patrón similar de declaración, aunque hay diversos métodos ayudantes dependiendo del tipo de relación.

Relaciones 1 a 1

Como ejemplo se tienen dos modelos relacionados: el modelo User debe informar que tiene un perfil asociado creado por el modelo Profile. Esta relación es muy común, porque realmente es muy necesario que dado un usuario se pueda acceder a su perfil.

Es tan sencillo como agregar este método a la clase User, definida en app/user.php

```
public function profile() {
  return $this->hasOne('App\Profile');
}
```

El patrón que se usa en esta y otros tipos de relaciones es crear un método con el nombre de aquel modelo que se esta relacionando. (`function profile()` para relacionar con el modelo Profile). Como código de la función se coloca un `return` sobre lo que te devuelve el método `hasOne()`, indicando a este método el modelo con el que se quiere relacionar con su namespace.

En esta ocasión el tipo de relación está definida por el método hasOne(), pero en otros casos se puede usar otros métodos como hasMany() o belongsTo().

Eloquent asume que la clave foránea de la tabla que relacionamos debe corresponder con la clave primaria "id" de la tabla de la que partimos (o aquel campo definido como $primaryKey). Si no es este el caso, se puede definir la relación indicando parámetros adicionales:

```
return $this->hasOne('App\Profile', 'clave_foranea',
'clave_local_a_relacionar');
```

También se podría definir en el modelo Profile la relación hacia el usuario al que pertenece.

Esta relación de pertenencia (un perfil pertenece a un usuario) se define mediante el método `belongsTo()`.

```
public function user()
{
  return $this->belongsTo('App\User');
}
```

El esquema de trabajo es muy similar. Se usa un método user, porque es el usuario el que se querrá relacionar al perfil. Luego se usa `belongsTo()` indicando el modelo con el que se esta relacionando.

Relaciones 1 a N:

Las relaciones de uno a muchos ocurren cuando tenemos dos tablas en la base de datos y en una de ellas se relaciona con la otra, de tal modo que la tabla "A" tiene muchos elementos de la tabla "B" relacionados y la tabla "B" sólo se relaciona con un elemento de la tabla "A".

Como ejemplo se tendrá dos modelos relacionados según sus tablas en la base de datos: el modelo Autor y el modelo Articulo tal que un articulo es escrito por un único autor, y que un autor puede escribir muchos artículos.

Para definir la relación tenemos que crear un método en el modelo, con el nombre que se quiere otorgar a dicha relación, que usualmente será el nombre de la entidad que se relacionara, en este caso en plural, dado que un escritor puede relacionarse con muchos artículos.

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;

class Autor extends Model
{
    public function articulos()
    {
        return $this->hasMany('App\Articulo');
    }
}
```

El método `articulos()` es el que implementa la relación. En él se debe devolver el valor de retorno del método `hasMany()` de los modelos Eloquent. A hasMany se le informara con el nombre de la clase del modelo con el que se esta relacionando.

Laravel entenderá automáticamente que en la tabla "articulos" existirá la clave foránea con el escritor que sea autor (autor_id en la tabla articulos de la BD) y que en la tabla local (autores), la clave primaria se llama "id". Es importante que se respeten las convenciones de nombrado de tablas y de claves foráneas, para que no se deba hacer más trabajo al definir las relaciones, pero si no es el caso, `hasMany()` también puede recibir como parámetros las personalizaciones en las tablas que sean necesarias.

Por ejemplo, si en la tabla articulos la clave foránea tuviera otro nombre distinto de autor_id, se indicara así:

```
return $this->hasMany('App\Articulo', 'nombre_clave_foranea');
```

Y si fuera el caso que en la tabla de autores la clave primaria no se llamara "id" también debera indicarlo con esta llamada a `hasMany()`:

```
return $this->hasMany('App\Articulo', 'nombre_clave_foranea',
'nombre_clave_primaria_local');
```

Una vez que se tiene en el modelo definida la relación, se accede a los datos de la tabla relacionada en cualquier modelo de Autor. Para ello se usa el nombre del método que fue creado como relación, en este caso era "articulos".

```
$articulos_de_un_autor = App\Autor::find(1)->articulos;
```

Esto será una colección de artículos a usar como cualquier otra collection de Laravel.

En Laravel el acceso a los datos de las tablas relacionadas se realiza por "lazy load", lo que quiere decir que, hasta que no se acceda a estos campos relacionados, no se hará la correspondiente consulta para la relación. Pero se puede forzar a Eloquent a que traiga de antemano los datos relacionados.

```
$escritores = Autor::with('articulos')->get();
```

La inversa de la relación 1 a N, en el modelo Articulo es definir en el modelo Articulo, si se ve necesario, la relación, para este ejemplo, acceder al autor que ha escrito un determinado artículo.

Esto se consigue con la definición de un método en el modelo, que llevará el nombre que queramos, pero usualmente será el nombre de la entidad que queremos relacionar. En este caso en singular, ya que solo se esta relacionando con un elemento de la otra tabla. En el caso es un artículo relacionado con un solo autor.

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Articulo extends Model
{
    public function autor()
    {
        return $this->belongsTo('App\Autor');
    }
}
```

El método que recibe la relación 1 a N inversa contiene la devolución del método `belongsTo()`, en el que se indica el nombre del modelo con el que sera relacionando.

Del mismo modo que en el caso anterior, es importante que las tablas estén creadas con las convenciones que asume Eloquent. En la tabla articulos se entiende que el nombre del campo de la clave foránea se llama "autor_id" y que el nombre de la clave primaria de la tabla relacionada es "id". Si no fuera el caso hay que aleccionar a Eloquent indicando como parámetros los nombres que se han utilizado en la definición de las tablas.

```
return $this->belongsTo('App\Autor', 'nombre_clave_foranea',
'nombre_clave_otra_tabla');
```

Las convenciones ahorran bastante trabajo al desarrollar los modelos, tanto colocando los nombres de las tablas en inglés y en plural, como usando "id" como clave primaria, y el nombre de la entidad seguido por "_" y luego "id" como clave foránea.

Relaciones N a M:

Las relaciones de muchos a muchos generan una tabla adicional, en la que se encuentran los índices de los dos elementos de la relación.

Por ejemplo, se usara la relación etiquetas y artículos. Una etiqueta puede tener asociados muchos artículos y un artículo puede tener asociadas varias etiquetas. En este caso tendremos, además de las tablas de "etiquetas" y "articulos", una tercera tabla que dará soporte a la relación de muchos a muchos. Ésta es la denominada "tabla pivote" o simplemente "pivot". Básicamente, la tabla pivot mantiene los identificadores de etiqueta y de articulo, para cada elemento con su relación.

El nombre simplemente es la unión de las dos entidades, separado por un guión bajo. Además, las entidades se ordenan alfabéticamente, por lo que la tabla se llamara: "articulos_etiquetas".

En la configuración de los modelos de Laravel, se colocara la relación con el método "`belongsToMany`".

En el modelo "Etiqueta" se debera crear un método que devuelva los artículos relacionados:

```
public function articulos() {
    return $this->belongsToMany('App\Articulo');
}
```

Para la relación inversa, desde Articulo a Etiqueta, usamos exactamente el mismo método "`belongsToMany`".

```
public function etiquetas() {
    return $this->belongsToMany('App\Etiqueta');
}
```

Una vez definida la relación, es posible acceder a los elementos relacionados sin tener que hacer consultas complicadas. Para ello se usara una propiedad en el modelo, que tiene el nombre igual al método que se ha usado para definir la relación.

Por ejemplo, si tengo una instancia de un modelo Articulo y se quiere acceder a sus etiquetas

```
$articulo->etiquetas
```

Esto permite acceder a una colección, que será el listado de todas las etiquetas que tiene el articulo que teníamos en el modelo.

Para obtener todas las etiquetas de varios articulos consultados:

```
$articulosConEtiquetas = Articulo::with('etiquetas')->get();
```

MÁS INFORMACIÓN

Para más información sobre como crear relaciones entre modelos, *eager loading*, etc. consultar directamente la documentación de Laravel en: https://laravel.com/docs/6.x/eloquent

CONSTRUCTOR DE CONSULTAS (*QUERY BUILDER*)

Laravel incluye una serie de clases que facilitan la construcción de consultas y otro tipo de operaciones con la base de datos. Además, al utilizar estas clases, se utiliza una notación mucho más legible, compatible con todos los tipos de bases de datos soportados por Laravel y que previene de cometer errores o de ataques por inyección de código SQL.

CONSULTAS

Para realizar un "Select" que devuelva todas las filas de una tabla usar el siguiente código:

```
$users = DB::table('users')->get();

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

En el ejemplo se utiliza el constructor `DB::tabla` indicando el nombre de la tabla sobre la que se va a realizar la consulta, y por último se llama al método `get()` para obtener todas las filas de la misma.

Si se desea obtener un solo elemento utilizar `first` en lugar de `get`, de la forma:

```
$user = DB::table('users')->first();

echo $user->name;
```

CLAUSULA *WHERE*

Para filtrar los datos se usara la clausula `where`, indicando el nombre de la columna y el valor a filtrar:

```
$user = DB::table('users')->where('name', 'Pedro')->get();

echo $user->name;
```

En este ejemplo, la clausula `where` filtrará todas las filas cuya columna `name` sea igual a `Pedro`. Si se desea realizar otro tipo de filtrados, como columnas que tengan un valor mayor (`>`), mayor o igual (`>=`), menor (`<`), menor o igual (`<=`), distinto del indicado (`<>`) o usar el operador `like`, indicarlo como segundo parámetro de la forma:

```
$users = DB::table('users')->where('votes', '>', 100)->get();

$users = DB::table('users')->where('status', '<>', 'active')->get();

$users = DB::table('users')->where('name', 'like', 'T%')->get();
```

Si se añaden más clausulas `where` a la consulta por defecto se unirán mediante el operador lógico `AND`. En caso de utilizar el operador lógico `OR` se tiene que usar `orWhere` de la forma:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'Pedro')
                    ->get();
```

orderBy / groupBy / having

También se pueden utilizar las clausulas `orderBy`, `groupBy` y `having` en las consultas:

```
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
                    ->get();
```

Joins

Si se requiere juntar dos o más tablas con Query Builder se usara el metodo `join()` invocado sobre el objeto Fluent Query Builder.

Este método recibe varios parámetros:

- La tabla a unir
- Campo de la tabla 1 por el que se relacionan
- Operador de la relación, usualmente "="
- Campo de la tabla 2 por el que se relacionan
- Tipo de join, por defecto "inner"
- Boolean, positivo para hacer un "join where", por defecto es false

En el siguiente ejemplo, dada una tabla de facturas y una de clientes, donde las facturas tienen un id_cliente.

```
$facturasCliente = DB::table('clientes')
        ->join('facturas', 'facturas.id_cliente', '=', 'clientes.id',
'inner', true)
        ->select('clientes.*', 'facturas.id as id_factura',
'facturas.fecha')
        ->where('clientes.email', '=', 'miguel@desarrolloweb.com')
        ->get();
```

Existen varios tipos de join y también varios métodos para hacer joins como `leftJoin()`, `rightJoin`, `joinWhere()`, etc. También existe una variante del método join que permite configurar la relación por medio de una función anónima. Para todos estos usos, consultar la documentación.

Inserts

Para insertar uno o varios campos en registros de una tabla se lanza el mensaje `insert()` a un objeto Fluent Query Builder.

```
$inserted = DB::table('books')
        ->insert([
                'name' => 'La Ciudad de los Prodigios',
```

```
                'author' => 'Eduardo Mendoza'
        ]);
```

La llamada al método `insert()` devuelve un boleano indicando si pudo completar la inserción con éxito.

Updates

Se Invoca el método `update()` sobre el Query Builder, al que se pasa un array asociativo con los datos a actualizar. Este comando se suele combinar con el método `Where`, para restringir el registro o grupo de registros que quieres actualizar con estos datos.

```
$updates = DB::table('books')
        ->where('id', '=', '74')
        ->update([
                'name' => 'Sin noticias de Gurb',
                'author' => 'Eduardo Mendoza'
        ]);
```

En este caso, como valor de retorno del `update` se obtiene el número de registros que se actualizaron al ejecutarse la consulta.

Como utilidad adicional, existen unos shortcuts para hacer una operación de incremento o decremento, aplicable en campos numéricos. En lugar de escribir a mano el código para hacer el update es realiza la llamada al método `increment()` o `decrement()`. El campo que se quiere incrementar y decrementar se envía como primer parámetro y de manera opcional se puede indicar un segundo parámetro con las unidades de incremento o decremento.

```
$num_updates = DB::table('books')
                ->where('id', '=', '97')
                ->increment('lecturas');
```

Deletes

Los deletes son exactamente iguales a los updates, solo que se tiene que invocar el método `delete()` para borrado.

```
$deletes = DB::table('facturas')->where('id', '=', '97')->delete();
```

Transacciones

Laravel también permite crear transacciones sobre un conjunto de operaciones:

```
DB::transaction(function()
{
    DB::table('users')->update(array('votes' => 1));

    DB::table('posts')->delete();
});
```

En caso de que se produzca cualquier excepción en las operaciones que se realizan en la transacción se desharían todos los cambios aplicados hasta ese momento de forma automática.

MÁS INFORMACIÓN

Para más información sobre la construcción de *Querys* (*join*, *insert*, *update*, *delete*, agregados, etc.) consultar la documentación de Laravel en su sitio web: https://laravel.com/docs/6.x/queries

LICENCIA

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar