



## 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 fichero de configuración **config/database.php** tenemos que indicar todos los parámetros de acceso a nuestras bases de datos y además especificar cuál es la conexión que se utilizará por defecto.

En Laravel podemos hacer uso de varias bases de datos a la vez, aunque sean de distinto tipo. Por defecto se accederá a la que especifiquemos en la configuración y si queremos acceder a otra conexión lo tendremos que indicar expresamente al realizar la consulta.

### *Configuración*

Lo primero que tenemos que hacer para trabajar con bases de datos es completar la configuración.

Si editamos el fichero con la configuración **config/database.php** podemos ver en primer lugar la siguiente línea: 'default' => env('DB\_CONNECTION', 'mysql'),

Este valor indica el tipo de base de datos a utilizar por defecto. El método env('DB\_CONNECTION', 'mysql') lo que hace es obtener el valor de la variable DB\_CONNECTION del **fichero .env**.

En el fichero de configuración **config/database.php**, dentro de la sección connections, podemos encontrar 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',
    'url' => env('DB_URL'),
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'laravel'),
    'username' => env('DB_USERNAME', 'root'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => env('DB_CHARSET', 'utf8mb4'),
```

```

        'collation' => env('DB_COLLATION', 'utf8mb4_unicode_ci'),
        'prefix' => '',
        'prefix_indexes' => true,
        'strict' => true,
        'engine' => null,
        'options' => extension_loaded('pdo_mysql') ? array_filter([
            (PHP_VERSION_ID >= 80500 ? \Pdo\Mysql::ATTR_SSL_CA :
\PDO::MYSQL_ATTR_SSL_CA) => env('MYSQL_ATTR_SSL_CA'),
        ]) : [],
    ],
],

```

Podríamos modificar directamente los valores del archivo config/database.php, pero cuando trabajamos en el desarrollo de una aplicación, la mayoría de las veces, **las credenciales de la base de datos de nuestro entorno local es diferente de las credenciales de nuestro entorno de producción o pruebas.**

Por lo que no es conveniente tener datos de configuración variables dentro de nuestro código.

Para solventar este problema podemos hacer uso de las variables de entorno **contenidas en el fichero .env**

Así que para configurar la base de datos vamos a cambiar las siguientes líneas en el **fichero .env de la raíz del proyecto:**

```

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=dwes_04_biblioteca_laravel
DB_USERNAME=root
DB_PASSWORD=

```

## *Migraciones: Schema Builder*

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.

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.

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

```
php artisan migrate
```

Para crear una nueva migración se utiliza el comando de Artisan `make:migration`, al cual le pasaremos el nombre del fichero a crear:

```
php artisan make:migration create_users_table
```

Esto nos creará un fichero de migración en la carpeta `database/migrations` con el nombre **`TIMESTAMP_create_users_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_votes_to_user_table --table=users
```

En este caso se creará también un fichero en la misma carpeta, con el nombre **`TIMESTAMP_add_votes_to_user_table.php`** preparado para modificar los campos de dicha tabla.

Por defecto, al indicar el nombre del fichero 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`**
- Si es una migración que modifica una tabla será **`actiontotable-name_table`**

Estructura de una migración

El fichero o clase PHP generada para una migración siempre contiene **los métodos `up` y `down`**.

- En el método `up` es donde tendremos crear o modificar la tabla
- En el método `down` tendremos que deshacer los cambios que se hagan en el `up` (eliminar la tabla o eliminar el campo que se haya añadido).

Esto nos permitirá poder ir añadiendo y eliminando cambios sobre la base de datos y tener un control o histórico de los mismos.

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 nos permitirá especificar la estructura de las tablas independientemente del sistema de base de datos que utilicemos.

## Crear y borrar una tabla

En **la sección up** para añadir una nueva tabla a la base de datos se utiliza el siguiente constructor:

```
public function up(): void
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
    });
}
```

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 utilizaremos para configurar las columnas de la tabla. En **la sección down** de la migración tendremos que eliminar la tabla que hemos creado, para esto usaremos el método:

```
public function down(): void
{
    Schema::dropIfExists('users');
}
```

Al crear una migración con el comando de Artisan **make:migration** ya nos 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). La columna id es la clave primaria de la tabla el tipo de dato es unsignedBigInteger. La columna timestamps posteriormente crea dos columnas en base de datos que son created\_at y updated\_at. Si queremos que nuestras tablas no tengan esas columnas debemos poner en su modelo \$timestamps=false;

## Añadir columnas

El constructor Schema::create recibe como segundo parámetro una función que nos permite especificar las columnas que va a tener dicha tabla. En esta función podemos ir añadiendo todos los campos que queramos, indicando para cada uno de ellos su tipo y nombre, y además si queremos también podremos indicar una serie de modificadores como valor por defecto, índices, etc. Ejemplo:

```
Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email')->unique();
    $table->timestamp('email_verified_at')->nullable();
    $table->string('password');
    $table->rememberToken();
    $table->foreignId('current_team_id')->nullable();
    $table->string('profile_photo_path', 2048)->nullable();
    $table->timestamps();
});
```

Podemos consultar todos los tipos de datos en <https://laravel.com/docs/master/migrations#creating-columns>

Añadir índices

Schema soporta los siguientes tipos de índices:

Comando	Descripción
<code>\$table-&gt;primary('id');</code>	Añadir una clave primaria
<code>\$table-&gt;primary(array('first','last'));</code>	Definir una clave primaria compuesta
<code>\$table-&gt;unique('email');</code>	Definir el campo como UNIQUE
<code>\$table-&gt;index('state');</code>	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 como figura en el ejemplo anterior al definir el campo email.

## Claves ajenas

Con Schema también podemos definir claves ajenas entre tablas:

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

// otra forma de definirlo

$table->foreignId('user_id')->constrained();
// más elegante aún

$table->foreignFor(User::class)->constrained();
```

En este ejemplo en primer lugar añadimos la columna “user\_id” de tipo UNSIGNED INTEGER (siempre tendremos que crear primero la columna sobre la que se va a aplicar la clave ajena).

A continuación creamos la clave ajena entre la columna “user\_id” y la columna “id” de la tabla “users”. También podemos 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 ajena, en el **método down** de la migración tenemos que utilizar el siguiente código:

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

Para indicar la clave ajena a eliminar tenemos que seguir el siguiente patrón para especificar el **nombre *tabla\_columna\_foreign*** donde:

- “tabla” es el nombre de la tabla actual
- “columna” el nombre de la columna sobre la que se creó la clave ajena.

## Ejecutar migraciones

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

```
php artisan migrate
```

Si nos aparece el error “class not found” lo podremos solucionar indicando a composer que vuelva a compilar el autocargador. Desde la carpeta del proyecto, ejecutamos este comando:

```
composer dump-autoload
```

El comando migrate 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 que queramos deshacer los últimos cambios podremos ejecutar:

```
php artisan migrate:rollback
```

O si queremos deshacer todas las migraciones:

```
php artisan migrate:reset
```

Un comando interesante cuando estamos desarrollando un nuevo sitio web es migrate:fresh, el cual deshacerá todos los cambios y volver a aplicar las migraciones:

```
php artisan migrate:fresh
```

Además si queremos comprobar el estado de las migraciones, para ver las que ya están instaladas y las que quedan pendientes, podemos ejecutar:

```
php artisan migrate:status
```

## *Modelos de datos mediante ORM*

El mapeado objeto-relacional (Object-Relational mapping o 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, podremos 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.** Para cada tabla de la base de datos tendremos que definir su correspondiente modelo, el cual se utilizará para interactuar desde código con la tabla.

### Definición de un modelo

Para definir un modelo que use Eloquent únicamente tenemos que crear una clase que herede de la clase Model. Podemos hacerlas “a mano”, pero es mucho más fácil y rápido crear los modelos usando el comando make:model de Artisan:

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

También podemos crear el modelo con una serie de flags como por ejemplo -mfs (m-modelo, f-factory s-seeder)

```
php artisan make:model Color -mfs
```

### Nombre del modelo

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.

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).

En el ejemplo anterior que hemos creado el modelo User buscará la tabla de la base de datos llamada users y en caso de no encontrarla dará un error. Si la tabla tuviese otro nombre lo podemos indicar usando la propiedad protegida \$table del modelo:

```
class User extends Model
```

```
{  
    protected $table = 'my_users';  
}
```

## Clave primaria

Laravel también asume que cada tabla tiene declarada una clave primaria con el nombre id. En el caso de que no sea así y queramos cambiarlo tendremos que sobrescribir el valor de la propiedad protegida \$primaryKey del modelo.

```
protected $primaryKey='my_id';
```

## Timestamps

Otra propiedad que en ocasiones tendremos que establecer son los timestamps automáticos. Por defecto Eloquent asume que todas las tablas contienen los campos updated\_at y created\_at (los cuales los podemos añadir 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 queramos utilizarlos (y que no estén añadidos a la tabla) tendremos que indicarlo en el modelo o de otra forma nos daría un error.

Para indicar que no los actualice automáticamente tendremos que modificar el valor de la propiedad pública \$timestamps a false, por ejemplo:

```
public $timestamps = false;
```

Ejemplo:

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

## Uso de un modelo de datos

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 indicar al inicio de la clase el espacio de nombres del modelo o modelos a utilizar. Por ejemplo, si vamos a usar los modelos User y Orders tendríamos que añadir: use App\User; use App\Orders;

## Consultar datos

Para obtener todas las filas de la tabla asociada a un modelo usaremos **el método all()**: Ejemplo:

```
$users = User::all();
foreach( $users as $user ) {
    echo $user->name;
}
```

Este método nos 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 podemos acceder a los campos o columnas de la tabla como si fueran propiedades del objeto (\$user->name).

También podremos 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);
```

Si queremos que se lance una excepción cuando no se encuentre un modelo podemos utilizar **los métodos findOrFail o firstOrFail**. Esto nos 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 con algunos de los métodos:

```
// 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();
```

También podemos 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 insertar un dato en una tabla de la base de datos tenemos que crear una **nueva instancia** de dicho modelo, **asignar los valores** y guardarlos con el **método save()**:

```
$user = new User();  
$user->name = "Juan";  
$user->save();
```

Para obtener el identificador asignado en la base de datos después de guardar, lo podremos recuperar accediendo al campo id del objeto que habíamos creado, por ejemplo:

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

## Actualizar datos

Para actualizar una instancia de un modelo sólo tendremos que recuperar la instancia que queremos actualizar, a continuación, modificarla y por último guardar los datos:

```
$user = User::find(1);  
$user->email = "prueba@correo.com";  
$user->save();
```

## Borrar datos

Para borrar fila de una tabla en la base de datos tenemos que usar su **método delete()**:

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

Si queremos borrar un conjunto de resultados también podemos usar el método delete():

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

## Operaciones con tablas relacionadas usando ORM

### Relaciones uno a uno

supongamos que tenemos dos modelos Libro y Portada, de modo que podemos establecer una relación de uno a uno entre ellos: un libro pertenece a una Portada y una portada pertenece a un libro.

Para reflejar esta relación en tablas, una de las dos debería tener una referencia a la otra. En este caso, podríamos tener un campo libro\_id en la tabla de portadas que indique a que libro pertenece esa portada. para indicar que un libro tiene una portada, añadimos un método en el modelo de Libro (portada), que se llame igual que el modelo con el que queremos conectar(Portada) que emplee el método **hasOne**.

```
class Libro extends Model
{
    public function portada(){
        return $this->hasOne(Portada::class);
    }
}
```

Ahora, si queremos obtener la portada de un libro, basta con que hagamos esto:

```
$portada=Libro::findOrFail($id)->portada;
```

La instrucción anterior obtiene el objeto Portada asociado con el libro buscado (a través del id del libro). Para que esta asociación tenga efecto, es preciso que en la tabla portadas exista un campo libro\_id y que corresponda con un campo id de la tabla de libros.

Si queremos utilizar otros campos distintos a los anteriores en una y otra tabla para conectarlas, debemos indicar dos parámetros más al llamar a hasOne.

Por ejemplo, así relacionaríamos las dos tablas anteriores, indicando que la clave ajena de portadas es idlibro, y que la clave de libros es codigo:

```
return $this->hasOne(Portada::class,'idlibro','codigo');
```

también es posible obtener la relación inversa; es decir, a partir de una portada, obtener el libro al que pertenece. Para ello añadimos un método en el modelo Portada y empleamos el método **belongsTo** para indicar a qué modelo se asocia:

```
class Portada extends Model
{
    public function libro(){
        return $this->belongsTo(Libro::class);
    }
}
```

De este modo, si quereemos obtener el Libro a partir de la portada, podemos hacerlo así:

```
$libro=Portada::findOrFail($idlibro)->libro;
```

## Relaciones uno a muchos

Supongamos que tenemos los modelos Autor y Libro, de modo que un autor puede tener varios libros, y un libro está asociado a un autor

La forma de establecer la relación entre ambos consistirá en añadir en la tabla libros una clave ajena al autor al que pertenece. A la hora de plasmar la relacion utilizamos el método **hasMany** en la clase Autor

```
class Autor extends Model
{
    public function libros(){
        return $this->hasMany(Libro::class);
    }
}
```

De este modo, obtenemos los libros asociados a un autor:

```
$libro=Autor::findOrFail($id)->libros;
```

También es posible establecer la relación inversa, y recuperar el autor al que pertenece un determinado libro, definiendo un método en la clase Libro que emplee **belongsTo**.

```
class Libro extends Model
{
    public function autor(){
        return $this->belongsTo(Autor::class);
    }
}
```

y obtener, por ejemplo, el nombre del autor a partir del libro:

## Relaciones muchos a muchos

Estas relaciones requieren contar con una tercera tabla que relacione las dos tablas afectadas. supongamos los modelos Libro y Biblioteca , de modo que un libro puede ser consultado en muchas bibliotecas, y una biblioteca tiene muchos libros. Nuevamente, definimos un método en el modelo Libro que utilice el método **belongsToMany** para indicar con qué otro modelo se relaciona:

```
class Libro extends Model
{
```

```

    public function bibliotecas(){
        return $this->belongsToMany(Biblioteca::class);
    }
}

```

así ya podremos acceder a las bibliotecas que tienen el libro:

```

$bibliotecas = Libro::findorFail($id)->roles;

```

En este caso, al otro lado de la relación hacemos lo mismo:

```

class Biblioteca extends Model
{
    public function libros(){
        return $this->belongsToMany(Libro::class);
    }
}

```

## Seeders

Los seeders sirven para rellenar la base de datos con datos iniciales. Para ello se puede rellenar **el método run** de database/seeds/DatabaseSeeder.php. Luego ejecutaremos la migración con:

```

php artisan db:seed

```

Pero lo habitual es crear distintos seeders para cada modelo de nuestra base de datos. Para crearlos usamos:

```

php artisan make:seeder UserSeeder

```

Nos creará **una clase con el método run**. Ahí escribiremos los datos de creación de objetos. Por último, desde el método run de la clase DatabaseSeeder llamaremos a las clases Seeder creadas:

```

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

```

## Ejemplo de Seeder de Libros

```

<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class BooksTableSeeder extends Seeder

```

```
{  
  /**  
   * Run the database seeds.  
   */  
  public function run(): void  
  {  
    DB::table('books')->insert(  
      [  
        'title' => '1984',  
        'published_year' => 1949,  
        'created_at' => now(),  
        'updated_at' => now(),  
        'author_id' => 2,  
        'isbn' => '978-0-452-28423-4',  
      ],  
    );  
  }  
}
```