Laravel: Database access & Models

DWES

0.- Introduction

Laravel makes it easy to set up and use different types of databases: MySQL, Postgres, SQLite, and SQL Server. The configuration file config/database.php indicates all the access parameters to our databases and, in addition, the connection that will be used by default. These same configuration parameters can be put in the .env file as seen previously.

In Laravel you can use several databases at the same time, even if they are of different types. By default, the one specified in the configuration will be accessed and if you want to access another connection, it will be expressly indicated when making the query.

1.- Configuration

In the config/database.php file we have the following:

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

```
'mysql' => [
    'driver' => 'mysql',
    'url' => env('DATABASE_URL'),
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
    'prefix_indexes' => true,
    'strict' => true,
    'engine' => null,
    'options' => extension_loaded('pdo_mysql') ? array_filter([
        PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
    ]) : [],
],
```

The variables that have the env() function are those that can be configured from the .env file

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

2.- Migrations

Migrations are a version control system for databases. They allow a team to work on a database by adding and modifying fields, maintaining a history of the changes made and the current state of the database. Migrations are used in conjunction with the Schema builder tool (discussed later) to manage the application's database schema.

The way migrations work is to create PHP files with the description of the table to be created and later, if you want to modify said table, a new migration would be added, a new PHP file, with the fields to modify.

Artisan includes commands to create migrations, to execute migrations or to rollback them.

The documentation is here: https://laravel.com/docs/12.x/migrations

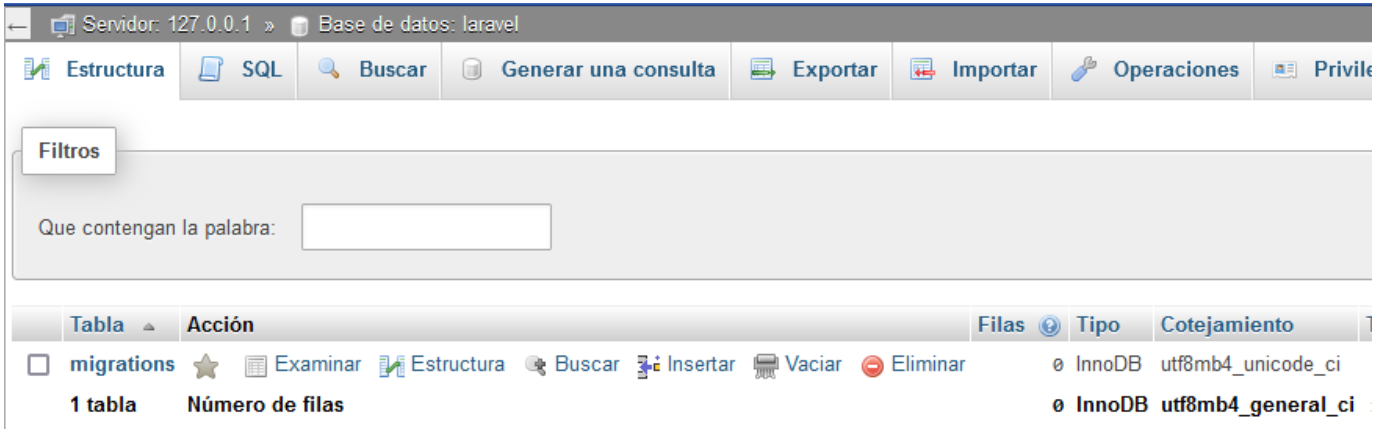
2.- Migrations

In order to start using the migrations you have to execute the command:

php artisan migrate:install

```
PS C:\xampp\htdocs\laravel\pruebadwes> php artisan migrate:install
Migration table created successfully.
```

If this command works correctly, a new table called migrations will be seen in the database from phpMyAdmin.



2.- Migrations

A migration class contains two methods: up and down.

- The up method is used to add new tables, columns, or indexes to your database. We will see the use of “Schema” later

- The down method should reverse the operations performed by the up method.

2.- Migrations

```php
<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }
```

2.- Migrations

```
    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::drop('flights');
    }
};
```

2.- Migrations

The Artisan command to **create a table** is the following:

```
php artisan make:migration NombreMigracion --create=NombreTabla
```

This will create a file with the name:

database/migrations/<TIMESTAMP>_NombreMigracion.php

To create a migration to **modify an existing table**, the command is:

```
php artisan make:migration NombreMigracion2 --table=NombreTabla
```

This will create a file with the name:

database/migrations/<TIMESTAMP>_NombreMigracion2.php

2.- Migrations

Although the name of the migrations is free and can be named whatever you want, it is recommended to use the following pattern:

- Creation: create_NombreTabla_table
  - Command: php artisan make:migration create_users_table --create=users
  - Generated file: database/migrations/<TIMESTAMP>_create_users_table.php

- Modification: Action_to_NombreTabla_table
  - Command: php artisan make:migration add_phonenumber_to_users_table --table=users
  - Generated file: database/migrations/<TIMESTAMP>_add_phonenumber_to_users_table.php

2.- Migrations

To run all of your migrations, execute the migrate Artisan command:

```
php artisan migrate
```

To see which migrations have been run, execute the status Artisan command:

```
php artisan migrate:status
```

If you would like to see the SQL statements that will be executed by the migrations without actually running them, you may provide the --pretend flag to the migrate command:

```
php artisan migrate --pretend
```

2.- Migrations

To roll back the latest migration operation, you may use the rollback Artisan command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

If you would like to see the SQL statements that will be executed by the migrations without actually running them, you may provide the --pretend flag to the migrate:rollback command:

```
php artisan migrate:rollback --pretend
```

The migrate:reset command will roll back **all** of your application's migrations:

```
php artisan migrate:reset
```

2.- Migrations

The migrate:refresh command will roll back all of your migrations and then execute the migrate command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh
```

The migrate:fresh command will drop all tables from the database and then execute the migrate command:

```
php artisan migrate:fresh
```

The main difference between them is “rollback” vs “drop”.

2.- Migrations

To create a new database table, use the create method on the Schema facade. The create method accepts two arguments:

- The name of the table
- A closure which receives a Blueprint object that may be used to define the new table

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

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

2.- Migrations

You may determine the existence of a table or column using the hasTable and hasColumn methods:

```
if (Schema::hasTable('users')) {
    // The "users" table exists...
}

if (Schema::hasColumn('users', 'email')) {
    // The "users" table exists and has an "email" column...
}
```

2.- Migrations

The table method on the Schema facade may be used to update existing tables. Like the create method, the table method accepts two arguments: the name of the table and a closure that receives a Blueprint instance you may use to add columns or indexes to the table:

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

2.- Migrations

Some supported types:

Command	Type
$table->boolean('confirmed')	BOOLEAN
$table->enum('difficulty', ['easy', 'hard'])	ENUM
$table->float('amount')	FLOAT
$table->increments('id')	Primary key: INTEGER auto increment
$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
$table->text('description')	TEXT

2.- Migrations

There are several column "modifiers" you may use when adding a column to a database table. For example, to make a column "nullable", you may use the nullable method:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

2.- Migrations

To **rename** an existing database table, use the rename method:

```php
use Illuminate\Support\Facades\Schema;

Schema::rename($from, $to);
```

To **drop** an existing table, you may use the drop or dropIfExists methods:

```php
Schema::drop('users');

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

2.- Migrations

For creating each type of **index** supported by Laravel use the following:

Command	Description
`$table->primary('id');`	Adds a primary key.
`$table->primary(['id', 'parent_id']);`	Adds composite keys.
`$table->unique('email');`	Adds a unique index.
`$table->index('state');`	Adds an index.
`$table->fullText('body');`	Adds a full text index (MySQL/ PostgreSQL).
`$table->fullText('body')->language('english');`	Adds a full text index of the specified language (PostgreSQL).
`$table->spatialIndex('location');`	Adds a spatial index (except SQLite).

2.- Migrations

To drop an index, you must specify the index's name:

Command	Description
`$table->dropPrimary('users_id_primary');`	Drop a primary key from the "users" table.
`$table->dropUnique('users_email_unique');`	Drop a unique index from the "users" table.
`$table->dropIndex('geo_state_index');`	Drop a basic index from the "geo" table.
`$table->dropFullText('posts_body_fulltext');`	Drop a full text index from the "posts" table.
`$table->dropSpatialIndex('geo_location_spatialindex');`	Drop a spatial index from the "geo" table (except SQLite).

2.- Migrations

Laravel also provides support for creating **foreign key constraints**, which are used to force referential integrity at the database level. For example, let's define a user_id column on the posts table that references the id column on a users table:

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

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

3.- Database seeding

Laravel includes the ability to seed your database with data using seed classes. Thanks to this option you can easily have test data.

To generate a seeder, execute the make:seeder Artisan command. All seeders generated by the framework will be placed in the database/seeders directory:

```
php artisan make:seeder UserSeeder
```

A seeder class only contains one method by default: run. This method is called when the db:seed Artisan command is executed. Within the run method, you may insert data into your database however you wish.

3.- Database seeding

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeders.
     */
    public function run(): void
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@example.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

3.- Database seeding

You may execute the db:seed Artisan command to seed your database. By default, the db:seed command runs the Database\Seeders\DatabaseSeeder class, which may in turn invoke other seed classes. However, you may use the --class option to specify a specific seeder class to run individually:

```
php artisan db:seed


php artisan db:seed --class=UserSeeder
```

3.- Database seeding

You may also seed your database using the migrate:fresh command in combination with the --seed option, which will drop all tables and re-run all of your migrations. This command is useful for completely re-building your database. The --seeder option may be used to specify a specific seeder to run:

```
php artisan migrate:fresh --seed


php artisan migrate:fresh --seed --seeder=UserSeeder
```

4.- Query Builder

Laravel's database query builder provides a convenient, fluent interface to creating and running database queries.

- Retrieving All Rows From a Table

You may use the table method provided by the DB facade to begin a query. The table method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally retrieve the results of the query using the get method:

```php
class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     */
    public function index(): View
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

4.- Query Builder

The get method returns an Illuminate\Support\Collection instance containing the results of the query where each result is an instance of the PHP stdClass object. You may access each **column's value** by accessing the column as a property of the object:

```php
use Illuminate\Support\Facades\DB;

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

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

4.- Query Builder

- Retrieving a Single Row / Column From a Table

If you just need to retrieve a single row from a database table, you may use the DB facade's first method. This method will return a single stdClass object:

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

return $user->email;
```

If you don't need an entire row, you may extract a single value from a record using the value method, and it will return the value of the column directly:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

4.- Query Builder

The query builder may also be used to add join clauses to your queries:

- Inner Join Clause
- Left Join / Right Join Clause
- Cross Join Clause (cartesian product)
- Advanced Join Clauses
- Subquery Joins

The query builder also provides a convenient method to "union" two or more queries together.

4.- Query Builder

Where clauses

You may use the query builder's where method to add "where" clauses to the query. The most basic call to the where method requires three arguments:

- The first argument is the name of the column
- The second argument is an operator, which can be any of the database's supported operators
- The third argument is the value to compare against the column's value

4.- Query Builder

For example, the following query retrieves users where the value of the votes column is equal to 100 and the value of the age column is greater than 35:

```
$users = DB::table('users')
                ->where('votes', '=', 100)
                ->where('age', '>', 35)
                ->get();
```

For convenience, if you want to verify that a column is = to a given value, you may pass the value as the second argument to the where method. Laravel will assume you would like to use the = operator:

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

4.- Query Builder

You may also pass an array of conditions to the where function. Each element of the array should be an array containing the three arguments typically passed to the where method:

```php
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

4.- Query Builder

You may use any operator that is supported by your database system:

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

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

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

4.- Query Builder

Or Where Clauses

When chaining together calls to the query builder's where method, the "where" clauses will be joined together using the and operator. However, you may use the orWhere method to join a clause to the query using the or operator. The orWhere method accepts the same arguments as the where method:

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

4.- Query Builder

If you need to group an "or" condition within parentheses, you may pass a closure as the first argument to the orWhere method:

```
$users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere(function (Builder $query) {
                $query->where('name', 'Abigail')
                      ->where('votes', '>', 50);
            })
            ->get();
```

The example above will produce the following SQL:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

4.- Query Builder

Where Not Clauses

The whereNot and orWhereNot methods may be used to negate a given group of query constraints. For example, the following query excludes products that are on clearance or which have a price that is less than ten:

```
$products = DB::table('products')
                ->whereNot(function (Builder $query) {
                    $query->where('clearance', true)
                          ->orWhere('price', '<', 10);
                })
                ->get();
```

4.- Query Builder

Additional Where Clauses

Laravel supports additional where clauses:

- whereBetween / orWhereBetween
- whereNotBetween / orWhereNotBetween
- whereBetweenColumns / whereNotBetweenColumns / orWhereBetweenColumns / orWhereNotBetweenColumns
- whereIn / whereNotIn / orWhereIn / orWhereNotIn
- whereNull / whereNotNull / orWhereNull / orWhereNotNull
- whereDate / whereMonth / whereDay / whereYear / whereTime
- whereColumn / orWhereColumn

4.- Query Builder

Advanced Where Clauses

The **whereExists** method allows you to write "where exists" SQL clauses. The whereExists method accepts a closure which will receive a query builder instance, allowing you to define the query that should be placed inside of the "exists" clause:

```
$users = DB::table('users')
           ->whereExists(function (Builder $query) {
               $query->select(DB::raw(1))
                     ->from('orders')
                     ->whereColumn('orders.user_id', 'users.id');
           })
           ->get();
```

4.- Query Builder

Alternatively, you may provide a query object to the whereExists method instead of a closure:

```
$orders = DB::table('orders')
                ->select(DB::raw(1))
                ->whereColumn('orders.user_id', 'users.id');

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

4.- Query Builder

Both of the examples will produce the following SQL:

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

4.- Query Builder

Sometimes you may need to construct a "where" clause that compares the results of a **subquery** to a given value. You may accomplish this by passing a closure and a value to the where method. For example, the following query will retrieve all users who have a recent "membership" of a given type:

```php
use App\Models\User;
use Illuminate\Database\Query\Builder;

$users = User::where(function (Builder $query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

4.- Query Builder

Or, you may need to construct a "where" clause that compares a column to the results of a subquery. You may accomplish this by passing a column, operator, and closure to the where method. For example, the following query will retrieve all income records where the amount is less than average:

```php
use App\Models\Income;
use Illuminate\Database\Query\Builder;

$incomes = Income::where('amount', '<', function (Builder $query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

4.- Query Builder

Ordering

- Order by

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

- Latest & oldest

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

- Random

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

4.- Query Builder

Grouping

- Group by & having

```php
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)
                ->get();
```

```php
$users = DB::table('users')
                ->groupBy('first_name', 'status')
                ->having('account_id', '>', 100)
                ->get();
```

- Group by & havingBetween

```php
$report = DB::table('orders')
                ->selectRaw('count(id) as number_of_orders, customer_id')
                ->groupBy('customer_id')
                ->havingBetween('number_of_orders', [5, 15])
                ->get();
```