Laravel: Migraciones

Hasta ahora, la creación y modificación de tablas y relaciones en una base de datos, las hemos realizado mediante código SQL, escrito en phpMyAdmin, o bien directamente en el código fuente de nuestra aplicación.

Las migraciones en Laravel, permiten crear y modificar el modelo físico de mi base de datos, a partir de una definición de su estructura, sin necesidad de utilizar SQL en mi aplicación o en phpMyAdmin.

Ventajas:

- Permiten sincronizar fácilmente las modificaciones de la base de datos en todos los entornos: Local de cada desarrollador, formación, pruebas, etc
- A la hora de crear la BD en producción, utilizar las migraciones asegura que el modelo creado es idéntico al que se utilizó durante el desarrollo
- Las migraciones son archivos en nuestra aplicación Laravel, susceptibles de subir a un repositorio Git, lo que nos permite controlar los cambios.
- Proporcionan documentación (en forma de código):
    - Sobre la estructura de la base de datos, en el fichero de migración
    - Sobre los cambios realizados en dicha estructura, en git

Forma de operar: Resumen

1. Generamos una migración con artisan, de forma automática. Podrá servir para crear una tabla, añadir o eliminar campos de la tabla, etc
2. Editamos el fichero generado, y completamos la especificación de cómo va a ser esa tabla, definimos los campos, etc
3. Ejecutamos con artisan las migraciones. Esto crea, de forma *física* en la base de datos, la estructura definida
4. Podemos “deshacer” (rollback) las migraciones ejecutadas, y la base de datos quedaría como estaba antes de realizar la migración.

Recomendación:

Durante un desarrollo con Laravel, se recomienda utilizar migraciones para generar nuestra base de datos física.
No se recomienda, *nunca*, utilizar SQL directamente en phpMyAdmin (o similar). El riesgo de generar inconsistencias entre distintos entornos es muy grande, y se pierde el control de los cambios.

La documentación de la versión 9 podemos encontrarla en:
https://laravel.com/docs/9.x/migrations

Configuración de entorno (environment):

Fichero .env

Uno de los ficheros que podemos encontrar en el raiz de nuestro proyecto Laravel es el .env, que contendrá nuestras variables de entorno, con información sensible (passwords de BD, URLs, etc).

Estas variables se cargan en la superglobal de PHP `$_ENV` cuando la aplicación recibe un request.

.env

```
...
# DB_CONNECTION=sqlite
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel1
DB_USERNAME=root
DB_PASSWORD=
...
```

Cuando creamos un proyecto con composer, este fichero se genera automáticamente, con unos valores por defecto, que deberemos modificar para adaptarlo a nuestra base de datos.

Este fichero, por seguridad, estará en **.gitignore**, nunca se llevará al repositorio de github.

Echemos un vistazo a nuestro .gitignore:

.gitignore

```
/node_modules
/public/build
/public/hot
/public/storage
/storage/*.key
/vendor
.env
.env.backup
.env.production
```

Tanto .env, como su versión de backup como su versión de producción, son información sensible de nuestro entorno, y nunca deberán estar en un repositorio compartido.

El manual de Laravel admite compartir este fichero en un repositorio, sólo en el caso de que vaya *encriptado*.

¿Cómo se utilizan las variables de entorno? Función env

Para recuperar el valor de estas variables de entorno, podemos utilizar la función env para recuperarlas y manipularlas:

```
env('DB_HOST', 'localhost');
```

- El primer parámetro es la variable de entorno que se está recuperando de .env
- El segundo parámetro es el valor por defecto de dicha variable. Si la variable no está definida, ese será el valor retornado por la función

En ningún fichero de configuración tendremos valores reales. Siempre se hará referencia a la variable de entorno correspondiente, que estará debidamente oculta y no compartida en .env

Más información sobre configuración:

https://laravel.com/docs/10.x/configuration

Configuración de la base de datos

La configuración de la conexión a la base de datos estará en el fichero:

```
config/database.php
```

Si nos fijamos, en el fichero no tenemos ni la URL de la base de datos, ni el usuario ni la password. Esa información es altamente sensible. Por eso utilizamos el valor de una variable de entorno, que lo guardará en el .env
En este fichero vienen por defecto configuraciones para conectarse a diversas bases de datos:

- sqlite: Manipula la información en ficheros, para aplicaciones muy pequeñas
- mysql: La tenemos instalada con el XAMPP, para aplicaciones más grandes. Para poder acceder desde Laravel, necesitamos iniciar el XAMPP.
- etc

Migraciones con Laravel

Llamamos migración de una BD a la creación de la estructura de la base de datos de la aplicación, en cualquier entorno. Laravel nos ofrece este servicio.

Lo primero que haremos será crear una base de datos nueva. Puedo crearla con phpMyAdmin, o acceder a mysql desde un terminal de la forma:

```
C:\xampp\mysql\bin> mysql -uroot -p  # En Windows
$ /opt/lampp/bin/mysql -uroot -p     # En Linux
```

Desde línea de comandos, se mostrará un prompt, podría crear una BD con create database:

```
MariaDB [(none)]> CREATE DATABASE martaBD;
```

La base de datos quedará creada.

También podemos utilizar phpmyadmin, u otro entorno gráfico para manejo de BD, y directamente seleccionar la opción de crear una nueva BD.

Ya no necesitaré hacer nada más desde el prompt de mysql, ni tampoco desde phpmyadmin. A partir de este momento, podré crear la estructura de mis datos desde Laravel.

Configuración de migración en Laravel:

Abro .env y modifico el nombre de la base de datos, el usuario, y la password:

.env

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

Una vez configurado, podré crear la estructura de la base de datos con el comando

```
$ php artisan migrate
```

Y obtendré el siguiente resultado:

Si vamos a phpMyAdmin, veremos que Laravel ha creado en la base de datos varias tablas nuevas:

- **users**: Una tabla para gestión de usuarios
- Otras tablas para control de passwords, tokens y errores, gestión que ofrece Laravel

Ha podido hacerlo, porque en la carpeta database/migrations existían varios ficheros php, uno por cada tabla creada, que indican cómo hacerlo. Estos ficheros vienen con Laravel.

- 0001_01_01_000000_create_users_table.php
- etc

En resumen: artisan migrations ejecutará los php que están en la carpeta migrations. Como había tres migraciones, ha creado todas las tablas definidas en dichas migraciones,

siguiendo el esquema indicado en dichos archivos. No olvidemos, que en .env he configurado el acceso a la base de datos.

Si editamos uno de los ficheros, veremos una función **up**, que crea la tabla. Para ello utiliza la clase **Schema** de Laravel, y su método create, que recibe como parámetro la función que crea las columnas. Utiliza para los tipos y las columnas la clase **Blueprint**, que tiene muchísimos métodos, dependiendo del tipo de datos de la columna que estemos creando.

También podríamos utilizar artisan migrate con muchas opciones si queremos modificar el comportamiento por defecto de migrate.

Esto es mejor que crear las tablas manualmente. Artisan creará todas las tablas que tengamos definidas en migrate. Definimos el esquema una vez, y lanzamos migrate siempre que lo necesitemos.

Creando y eliminando migraciones en Laravel

Creación de una migración para crear una tabla

Vamos a crear una migración desde cero, para la creación de una tabla llamada posts

```
$ php artisan make:migration create_posts_table --create="posts"
```

El comando anterior indica:

1. **make:migration**: Vamos a crear una migracion
2. **create_posts_table**: Este es el nombre de la migración (seguiremos la convención de nombres de Laravel)
3. **--create="posts"**: Este flag indica que la migración creará una tabla llamada posts

Efectivamente, ha creado el fichero php en migrations, con el nombre que hemos indicado, precedido por el año_mes_dia_hhmmss_ en que es creado el fichero.

El contenido del fichero, por defecto, es el siguiente:

2022_12_30_122858_create_posts_table.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()
    {
```

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

    /**
     * Reverse the migrations.
     */
    public function down()
    {
        Schema::dropIfExists('posts');
    }
};
```

Por defecto añade a la tabla dos columnas, pero yo tendré que indicar el resto de la estructura, añadiendo mis campos en el método up.

En mi caso voy a crear una tabla de posts, con título y cuerpo del post:

```
public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('titulo');
            $table->text('cuerpo');
            $table->timestamps();
        });
    }
```

Si queremos campos únicos, o con valor por defecto, así como el resto de características de los campos, se haría de la siguiente forma:

- `$table->string('titulo')->unique();`
- `$table->text('cuerpo')->default('eee');`
- etc (ver manual)

Una vez hemos creado nuestra migración.

Para crear la tabla en la base de datos, bastaría ejecutar:

```
$ php artisan migrate
```

Y quedaría creada en la base de datos:

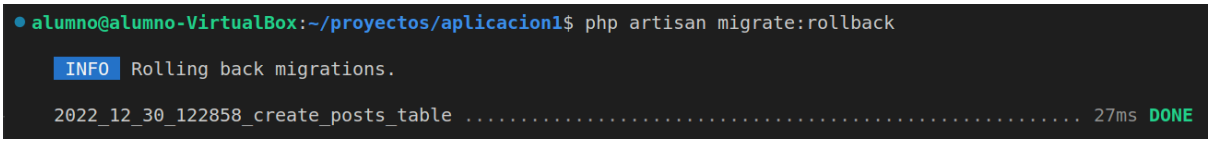
Podemos comprobarlo en phpmyadmin.

Rollback de la migración realizada

Imaginemos que queremos deshacer lo que acabamos de crear, que no queremos que el campo se llame cuerpo sino contenido. Bastaría ejecutar el comando:

```
$ php artisan migrate:rollback
```

La respuesta sería:



Si vamos a phpmyadmin, veremos que la tabla ha desaparecido. Lo que ha hecho Laravel es deshacer la migración, ejecutando la función **down.**

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

Una vez hecho el rollback, puedo modificar la migración, cambio la estructura de la tabla en el **up** de la migración, y vuelvo a ejecutar **php artisan migrate** para que coja los cambios.

Añadiendo columnas a tablas ya existentes

Imaginemos que queremos añadir columnas a una tabla, y no queremos borrarla porque contiene datos.

En este caso, podemos hacerlo con migrate, añadiendo una nueva migración que añada una columna.:

```
$ php artisan make:migration add_is_admin_column_to_posts_table --table=posts
```

Esto creará una nueva migración, esta vez la función up no utiliza Schema::create, sino Schema::table, lo que significa que modificará la tabla como yo le indique.

Añadiremos el siguiente código para añadir la columna is_admin a la tabla posts:

```
public function up()
    {
        Schema::table('posts', function (Blueprint $table) {
            //
            $table->integer('is_admin')->unsigned();
        });
    }
```

```php
    /**
     * Reverse the migrations.
     */
    public function down()
    {
        Schema::table('posts', function (Blueprint $table) {
            //
            $table->dropColumn('is_admin');
        });
    }
```

Nota: En este caso, la columna a añadir es un entero sin signo.

Una vez preparada la migración, sólo tendré que ejecutar migrate:

```
$ php artisan migrate
```

Este comando añadirá la nueva columna a la tabla.

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan migrate

  INFO  Running migrations.

  2022_12_30_165410_add_is_admin_column_to_posts_table ..... 36ms DONE
```

Voy a phpMyAdmin, y veo que ha añadido la columna:

6	**is_admin**	int(10)	UNSIGNED

Imaginemos que queremos que la columna sea un entero más pequeño. En Laravel tenemos el tipo tinyInteger, vamos a utilizarlo..

Lo ideal es seguir los pasos de rollback:

1. Ejecuto un rollback´, que ejecutará el **down** eliminando la columna is_admin de la tabla:
   ```
   $ php artisan migrate:rollback
   ```
2. Modifico la migración para cambiar la estructura de la columna de la forma:
   ```
   $table->tinyInteger('is_admin')->unsigned();
   ```
3. Vuelvo a ejecutar migrate para que la cree de nuevo:
   ```
   php artisan migrate
   ```

Si voy a comprobar phpMyAdmin y refresco, veré que ahora el campo es:

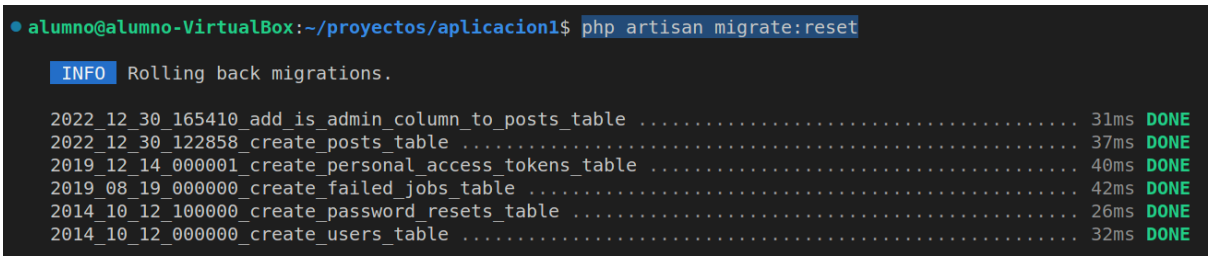
6	**is_admin**	tinyint(3)	UNSIGNED

Algunos comandos más para migraciones

Comando reset

Elimina todas las tablas que hayamos hecho hasta ahora, incluidas las que vienen con las migraciones de Laravel. Es como hacer un rollback de todas las migraciones. No borra los ficheros php de migración, sólo ejecuta su función **down**.

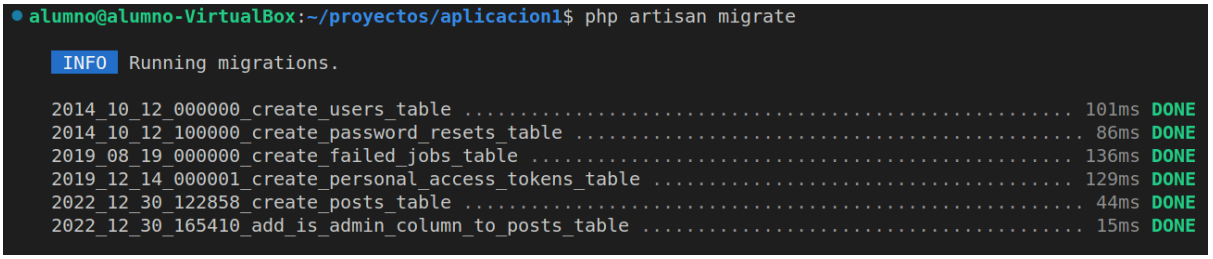
```
$ php artisan migrate:reset
```



Este comando deja en la base de datos la tabla migrations para que podamos hacer un seguimiento de todas las migraciones realizadas.

Comando migrate

Ejecuta los **up** de todos los ficheros de migración, con el resultado de crear la estructura (vacía) que haya definido hasta el momento.



Comando refresh

Si hacemos varios cambios en los esquemas de varias migraciones, podemos, en vez de hacer rollback y volver a lanzar migration manualmente, lanzar el siguiente comando:

```
$ php artisan migrate:refresh
```

Este comando realizará un rollback y un migrate de todas las migraciones, actualizando toda la información de nuestra base de datos.

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan migrate:refresh

  INFO  Rolling back migrations.

  2022_12_30_165410_add_is_admin_column_to_posts_table ..... 15ms DONE
  2022_12_30_122858_create_posts_table ..... 27ms DONE
  2019_12_14_000001_create_personal_access_tokens_table ..... 27ms DONE
  2019_08_19_000000_create_failed_jobs_table ..... 33ms DONE
  2014_10_12_100000_create_password_resets_table ..... 28ms DONE
  2014_10_12_000000_create_users_table ..... 27ms DONE

  INFO  Running migrations.

  2014_10_12_000000_create_users_table ..... 91ms DONE
  2014_10_12_100000_create_password_resets_table ..... 84ms DONE
  2019_08_19_000000_create_failed_jobs_table ..... 84ms DONE
  2019_12_14_000001_create_personal_access_tokens_table ..... 203ms DONE
  2022_12_30_122858_create_posts_table ..... 44ms DONE
  2022_12_30_165410_add_is_admin_column_to_posts_table ..... 35ms DONE
```

Comando status

Este comando indica el estado de las migraciones: Si han sido ejecutadas (Ran) o si no (Pending).

```
$ php artisan migrate:status
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan migrate:status

  Migration name ..... Batch / Status
  2014_10_12_000000_create_users_table ..... Pending
  2014_10_12_100000_create_password_resets_table ..... Pending
  2019_08_19_000000_create_failed_jobs_table ..... Pending
  2019_12_14_000001_create_personal_access_tokens_table ..... Pending
  2022_12_30_122858_create_posts_table ..... Pending
  2022_12_30_165410_add_is_admin_column_to_posts_table ..... Pending
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan migrate:status

  Migration name ..... Batch / Status
  2014_10_12_000000_create_users_table ..... [1] Ran
  2014_10_12_100000_create_password_resets_table ..... [1] Ran
  2019_08_19_000000_create_failed_jobs_table ..... [1] Ran
  2019_12_14_000001_create_personal_access_tokens_table ..... [1] Ran
  2022_12_30_122858_create_posts_table ..... [1] Ran
  2022_12_30_165410_add_is_admin_column_to_posts_table ..... [1] Ran
```