DWES UD6 Apuntes

1. Migraciones

→ Elemento de Laravel para crear/modificar tablas en la bd.

→ **Métodos:**

- up() - crea/modifica el esquema de la tabla
- down() - elimina/deshace la tabla

Si trabajamos con bd relacional, se pueden establecer las relaciones entre las tablas.

→ Crear una migración:

Opción 1: crear archivo dentro de la carpeta migrations con esta estructura:
yyyy_mm_dd_hhmmss _create/update_to_nombre_tabla_table

Opción 2: con el comando artisan:
php artisan make:migration create_customers_table

- softDeletes() - función que se añade cuando se crean las tablas para que se cree la columna "delete_at", que permite eliminar un registro pero que se guarde su registro para poder recuperarlo. De esta forma, cuando Laravel liste los datos desde la bd, obviará aquellos registros que contengan valor en la columna "delete_at".
- update_to – si la migración es un update_to(), no crea un esquema, sino que se usa para añadir campos al esquema de creación:

php artisan make:migration update_to_customers_table
→ Se añadirá el tipo de dato y el nombre de lacolumna.

- after() - se usa junto con update_to() en caso de querer posicionar la nueva columna en un orden en concreto. Se le pasa como param la columna que la precede.

→ Tras haber creado las migraciones, se pueden crear las tablas en la bd. Con el siguiente comando, se generan/modifican todas las tablas correspondientes a cada migración:
php artisan migrate

Además, se genera una tabla ***migrations*** que registra un histórico de migraciones.
Esta tabla tiene la columna ***batch*** con info sobre las ejecuciones del ***php artisan migrate.***
Este sistema permite eliminar las últimas migraciones realizadas usando el comando:
php artisan migrate:rollback

- artisan migrate:refresh – añade todas las migraciones en un único bloque y se registran en un único batch, eliminando el contenido de todas las tablas y volviendo a generarlo.

2. El ORM Eloquent de Laravel

Se pueden realizar operaciones con la bd mediante el uso de Eloquent, proporcionado por el framework.

Permite centralizar las peticiones referentes a cada tabla en una clase distinta llamada modelo, pudiendo hacer uso de los objetos del modelo para realizar las operaciones con la bd.

Modelos

Se utiliza un modelo para cada tabla de la bd (menos para las tablas pivote, que se crean de manera excepcional).

Los modelos tienen el mismo nombre que las tablas pero en SINGULAR y empezando por mayuscula (como una clase de php) y PascalCase y en inglés.

→ Crear un modelo:

Opción 1: crear archivo dentro de app/models.

Opción 2: con el comando artisan:

```
php artisan make:model Nombre_del_modelo
```

→ También se puede crear el modelo + hacer la migración usando -m:

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

La clase Model:

→ Cada modelo será una clase que extenderá de la clase Model de Eloquent, por lo que tiene métodos propios.

→ Algunos de los atributos de la clase Model se pueden modificar:

- protected $table – si modificamos este atributo, se nombra al modelo de diferente forma que la tabla de la que depende.

→ Uso del trait añadiendo dentro de la clase la instrucción “use SoftDeletes;” e importándolo con “use Illuminate\Database\Eloquent\softDeletes (previamente configurado).

Relaciones con Eloquent ORM

Se permite a nivel de código establecer los tipos de relaciones entre modelos.
Las relaciones permiten obtener objetos de un modelo usando otro modelo.
Para poder definir estas relaciones existen diferentes métodos para establecer las relaciones entre los modelos, dependiendo del tipo de relación que tengan:

OTO – Uno a Uno

→ Se necesita saber cuál es el modelo principal (modelo A) y cuál es el que contendrá la FK (modelo B).

→ En el modelo A, se crea un método llamado “modeloB” que contiene el método de Model hasOne() y contiene como param la clase modeloB:

```
class modeloA extends Model {
        public function modeloB() {
                return $this→hasOne(ModeloB::class);
        }
}
```

→ En el modelo B, se crea un método llamado “modeloA” que contiene el método de Model belongsTo() y contiene como param la clase modeloA:

```
class modeloB extends Model {
        public function modeloA() {
                return $this→belongsTo(ModeloA::class);
        }
}
```

OTM – Uno a Muchos

→ Se necesita saber cuál es el modelo principal (modelo A) y cuál es el que contendrá la FK (modelo B).

→ La función del modelo principal debe tener una función que devuelva múltiples instancias del modelo que contiene la FK.

→ En el modelo A, se crea un método llamado “modeloB” que contiene el método de Model hasMany() y contiene como param la clase modeloB:

```
class modeloA extends Model {
        public function modeloB() {
                return $this→hasMany(ModeloB::class);
        }
}
```

→ En el modelo B, se crea un método llamado “modeloA” que contiene el método de Model belongsTo() y contiene como param la clase modeloA:

```
class modeloB extends Model {
        public function modeloA() {
                return $this→belongsTo(ModeloA::class);
        }
}
```

OJO!! Para poder ejecutar la relación desde el modeloB, se debe haber creado previamente el modelo y la migración de éste (“php artisan make:model ModeloB -m”).

→ También se pueden concatener otros métodos, dependiendo de la información que queramos obtener:

- latestOfMany() - se obtiene el último valor.
- oldestOfMany() - se obtiene el valor más viejo.

- ofMany() - se obtiene el valor mínimo/máximo de una determinada columna.

(HOT) & (HTM) – Tienen Uno y Tiene Muchos

→ HOT = relación OTO con otro modelo a través de un modelo intermedio

→ HTM = relación HTM con otro modelo a través de un modelo intermedio

→ Tendremos 3 modelos involucrados, siendo el modeloB el que corresponde a la tabla puente entre ambos.

→ Para establecer la relación entre el modeloA y el modeloC, hay que crear un método interno en el modeloA con el nombre modeloC(). Dentro de éste, se llamará al método de la clase Model hasManyThrough() que recibe los siguientes params:

1º - nombre del modelo que se desea obtener (modeloC)

2º - nombre del modelo que hace de intermediario (modeloB)

Resto de params, serían las PK y FK de los modelos:

3º - FK en modeloC

4º - FK en modeloC

5º - PK en modeloA

6º - PK en modeloB

```
class modeloA extends Model {
        public function modeloC() {
                return $this→hasManyThrough(ModeloC::class, ModeloB::class);
        }
}
```

(MTM) – Muchos a Muchos

→ Se necesita de una tabla pivote que contenga entre sus columnas la PK de cada modelo como NOT NULL

→ La tablaPivote se llamará tablaX_tablaXX (en orden alfabético).

→ Si dentro de la tablaPivote existen columnas adicionales, se puede hacer uso del método withPivot(), donde se le pasan las diferentes columnas separadas por comas.

→ Tendremos modeloA y modeloB involucrados, de modo que:

<u>En el modeloA:</u>

→ modeloA tendrá un método interno modeloB() que llamará al método de la clase Model belongsToMany(), que recibe los siguientes params:

1º – nombre del modeloB

2º – tablaPivote
3º – PK del modeloA
4º – PK del modeloB

En el modeloB:
→ modeloB tendrá un método interno modeloA() que llamará al método de la clase Model belongsToMany(), que recibe los siguientes params:
1º – nombre del modeloA
2º – tablaPivote
3º – PK del modeloB
4º – PK del modeloA

```
class modeloA extends Model {
        public function modeloB() {
                return $this→belongsToMany(modeloB::class)->withPivot('column_name');
        }
}
```

*** También se pueden añadir las columnas timestamps de forma automática concatenando la llamada al métoco withTimestamps().

```
class modelo extends Model {
        public function modeloA() {
                return $this→belongsToMany(modeloA::class)->withTimestamps();
        }
}
```

***No es común, pero se pueden usar modelos para referenciar las tablas pivote de la siguiente manera:
Se llama al método using() concatenado que recibe como param el nombre del modelo que debe usar.

```
        return $this->belongsToMany(modeloB::class)->using(modeloPivote::class);
```

OJO!!

→ Para poder ejecutar la relación desde el modeloB, se debe haber creado previamente el modelo y la migración de éste ("php artisan make:model ModeloB -m").

→ Se necesita también la migración de la tablaPivote, que debe contar con las PK de las otras 2 tablas + la suya propia:

```
        php artisan make:migration create_pivot_table
```

Relaciones polimórficas

→ Cuando un modelo pertenece a otros modelos y es necesario implementar como FK las PK de esos otros modelos, únicamente se usará como FK una única PK de los otros modelos, teniendo que ser NULL el resto de ellas.

→ Si este modelo pertenece a muchos otros modelos, el n.º de columnas puede aumentar haciendo poco mantenible la bd, pues tendrá mucha columnas a NULL y sólo rellenará la columna de la FK a la que pertenezca.

→ Solución a la casuística anterior: las RELACIONES POLIMÓRFICAS

→ No se usa una columna por cada FK; Se crean sólo 2 columnas (ej. usa el modelo ***Image***:

1. nombre del modelo con el sufijo -able + id ***imageable_id***
2. nombre del modelo con el sufijo -able + type ***imageable_type***

→ Este modelo almacenará en la bd el id del modelo al que pertenece la imagen dentro de la columna imageable_id y el nombre del modelo padre que la crea en la columna imageable_type.

Métodos en relaciones polimórficas:

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

OJO!! En los métodos belongsTo(), hasOne(), hasOneThrough() y morphOne() se permite definir un modelo predeterminado que se devolverá si la relación es nula.

→ Tras llamar al método, se añade →withDefault(), que puede recibir un array asociativo con los atributos del modelo que se espera obtener y los valores predeterminados para estos atributos.

```php
class Image extends Model
{
    /**
     * Obtiene el modelo propietario de la imagen.
     */
    public function imageable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Obtiene la imagen del perfil del cliente.
     */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}

class User extends Model
{
    /**
     * Obtiene las imágenes de la dirección.
     */
    public function image()
    {
        return $this->morphMany(Image::class, 'imageable');
    }
}
```