BD - RELACIONES CON ELOQUENT

La ventaja de utilizar Eloquent, es que implementa de forma automática las relaciones.

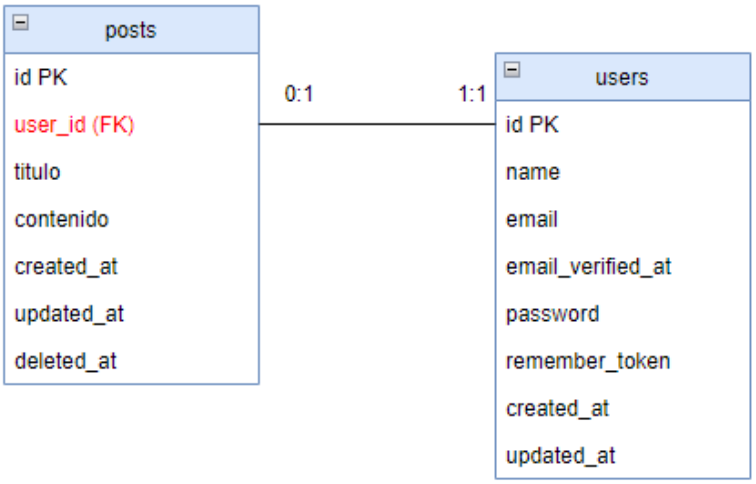
En el ejemplo de las transparencias anteriores, si queremos relacionar los post con el usuario, de forma que un usuario pueda tener varios posts, tendríamos una relación 1 a n entre usuarios y posts.

Para recuperar los posts de cada usuario, basta con hacer una función en el modelo Post.php, y con una única línea de código, utilizando una función Eloquent, recupera todos los posts del usuario.

A continuación, vamos a ver cómo se implementan las relaciones en Eloquent, y lo sencillo que resulta el código para manipularlas.

Relaciones 1 a 1 (recuperar el hijo único)

Imaginemos una relación 1 a 1 entre usuarios y posts: Un usuario tendrá un único post, y un post pertenecerá a un único usuario:



Para implementar este modelo con Eloquent, necesitaremos realizar los siguientes pasos:

1. Modificamos la migración de la tabla posts para añadir una columna con la identificación del usuario del post (user_id). Esa relación será 1 a 1, funcionalmente hablando. Un usuario tendrá un único post.
2. Creamos un usuario en la tabla users.
3. Añadimos un post, y pondremos el id del usuario creado anteriormente en el campo user_id. Con esto, el post queda ligado al usuario.
4. Modificamos el modelo de la tabla users (User.php), creando una función pública post, que retornará el post del usuario con la función hasOne. Laravel tratará esta función como si fuera un atributo, de forma que para referirnos a ella en el futuro no pondremos paréntesis.

1.- 2022_12_30_122858_create_posts_table.php

```
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->unsignedBigInteger('user_id');
            $table->string('titulo');
            $table->text('contenido');
            $table->timestamps();
            $table->softDeletes();

            // Foreign key físico en BD, si queremos añadirlo
            $table->foreign('user_id')->references('id')->on('users');
        });
```

Para coger los cambios, tenemos que refrescar:

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

4.- User.php

```
    /**
     * Implementa la relación 1 a 1: Un usuario un post
     */
    public function post() {
        return $this->hasOne('App\Models\Post');
    }
```

En este caso le estamos facilitando encontrar el modelo de la tabla posts, incluyendo su espacio de nombres.
La función hasOne, busca un post en la tabla posts, cuyo user_id sea igual al de $this.
Si el nombre de la foreign key en la tabla posts no es “user_id”, en ese caso tendremos que añadir un segundo parámetro a hasOne para indicar el nombre de la foreign key:

```
hasOne(‘App\Models\Post’,’id_de_usuario’)
```

Además, Eloquent asume que la clave foránea de posts coincidirá con el id del registro de usuario actual. Si no queremos utilizar ese valor, podríamos añadir un tercer parámetro, indicando cuál utilizar.

<u>Uso de la relación:</u>
Una vez definida la relación, vamos a web.php y añadimos una ruta para recuperar el post de un usuario:

web.php

```
use App\Models\User;
Route::get('/user/{id}/post', function($id) {
    return User::find($id)->post;
})
```

La variable id me permite seleccionar el código del usuario. find se aplica a ese id, es decir, encontrará el usuario cuyo id se pone en la propia ruta.
En cuanto al método post, es la función pública que hemos añadido al modelo de User.php. Recupera el post único del usuario con la función hasOne.
Nota: Tenemos que incluir la clase User (modelo de la tabla users).

El resultado, en formato JSON, es el siguiente:

Como puede verse, se muestra la información del post correspondiente al usuario 1.

Podemos obtener la información de forma más específica concatenando atributos y métodos de la clase User. Por ejemplo, retornar únicamente el título del post con:

```
return User::find($id)->post->titulo;
```

Inversa de una relación uno a uno (recuperar el padre)

En el ejemplo anterior, ¿cómo obtenemos el usuario de un post ?:

1. Creamos en Post.php (el modelo de post) la funcionalidad para obtener el usuario, en este caso un método user
2. Creamos la ruta para que retorne la información del usuario asociado al post indicado en dicha ruta

Post.php

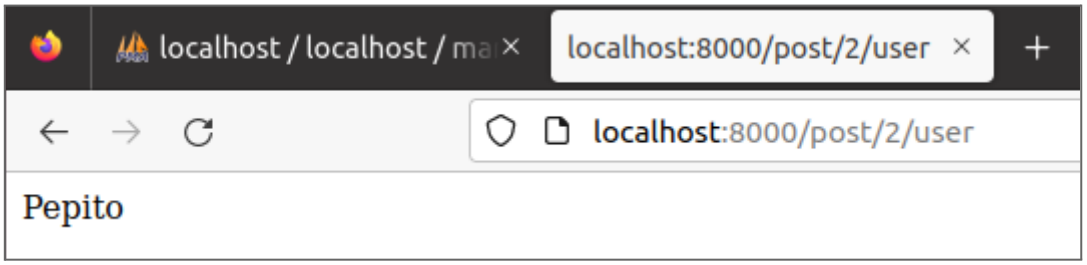
```
    public function user() {
        return $this->belongsTo('App\Models\User');
    }
```

La función belongsTo permite recuperar el “padre” en la relación. En el ejemplo, un post es el hijo, y belongsTo retorna la información del padre, en este caso el usuario. El parámetro que recibe es la clase modelo del padre, en nuestro caso User, con su espacio de nombres.

web.php

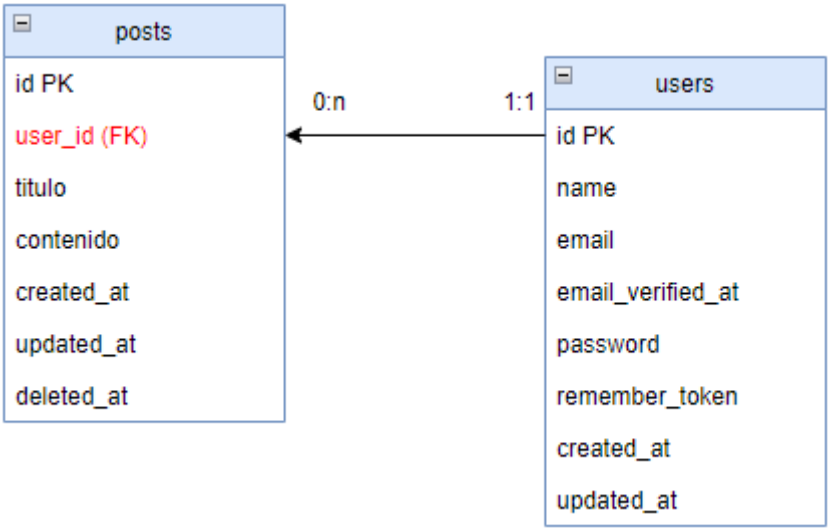
```
// Inversa de una relación - Obtener el usuario de un post (el padre)
Route::get('/post/{id}/user', function($id) {
    return Post::find($id)->user->name;
})
```

En el siguiente ejemplo, tenemos un post con id=2, y su usuario es Pepito:



Relación 1 a N

Vamos a modificar la relación anterior. En este caso, vamos a permitir que un usuario tenga varios posts.



En este caso, para implementar la relación directa:

1. Creamos una ruta que obtenga todos los posts de un usuario dado (en web.php)
2. Creamos en el modelo de User una función posts que retorne todos los posts del usuario. Esta función será tratada como una propiedad, un atributo, por lo que no utilizaremos paréntesis cuando la utilicemos.

User.php

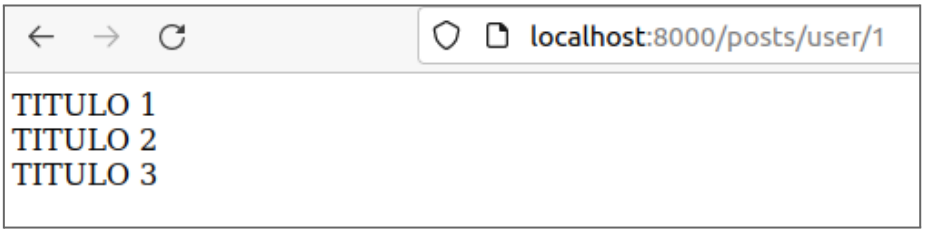
```
    public function posts() {
        return $this->hasMany('App\Models\Post');
    }
```

La función hasMany retornaría todos los posts correspondientes al usuario $this, en un array de objetos. Cada objeto será un registro de la tabla posts.

web.php

```
// Relación uno a muchos
Route::get('/posts/user/{id}', function($id) {
    $user = User::find($id);
    foreach ($user->posts as $post) {
        echo $post->titulo . '<br>';
    }
})
```

Ponemos echo, porque no podemos retornar con return varias veces, sólo cogería el primero.
Si por ejemplo, elegimos el usuario 1, y tiene tres posts asociados en la tabla posts, el resultado sería:



Actualización del modelo de datos

Con Eloquent, no necesitamos ir a phpMyAdmin y actualizar la estructura de las tablas cada vez que haya un cambio.

Lo más eficiente y sencillo es:

1. Escribo las migraciones en Laravel. Toda la estructura de la base de datos se definirá en ellas.
2. Vacío la base de datos eliminando todas las tablas
3. Hago php artisan migrate
4. Ahora todas las tablas tienen la estructura definida en el modelo de Laravel, y cuando reinstale el modelo en distintos ordenadores, será fidedigno con la definición de mi aplicación.

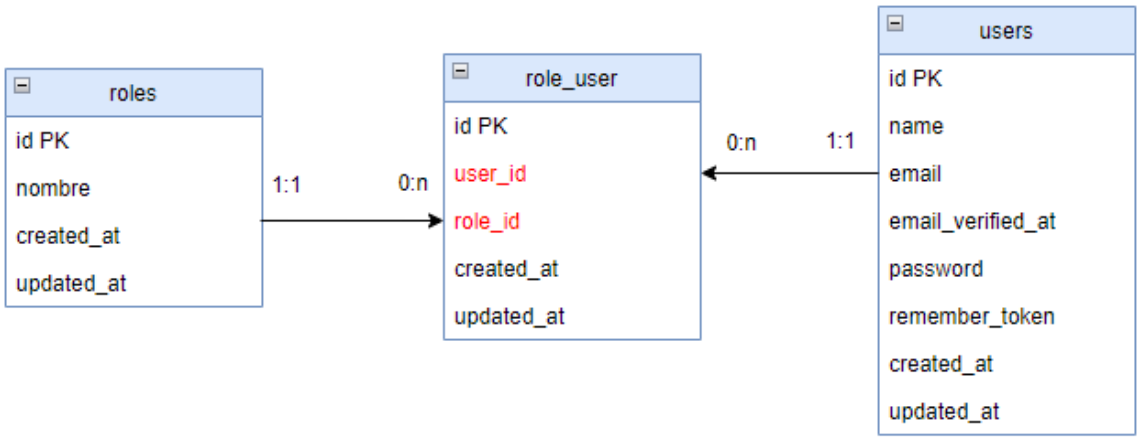
Ventajas:

Esta forma de trabajar hace que el modelo de mis programas coincida siempre con el modelo implementado en la base de datos, ya que la definición está integrada con el código de la aplicación.

Por otra parte, siempre que queramos crear una nueva base de datos en un nuevo entorno (local, pruebas, preproducción, etc), podemos estar seguros de que será el mismo que utilizamos en la aplicación. Además, quedará desplegado con un único comando: migrate

RELACIÓN N A N

Cuando tenemos una relación “muchos a muchos”, por ejemplo, que un usuario tenga varios roles, y que un rol pueda ser desempeñado por varios usuarios. En este caso, se normaliza la relación con una tabla intermedia de relación, que en inglés se llama tabla “pivot” (tabla pivote). Por tanto tendremos la siguiente estructura:



Para ilustrar esta relación crearemos la siguiente estructura:

1. Un modelo para la tabla **roles**, con su migración. Luego actualizaremos la migración para añadir los campos necesarios.
2. Una migración para la tabla de relación **role_user**, la convención es que la llamemos role_user, en singular, así Laravel la manejará automáticamente como tabla de relación, no tendremos que manejarla nosotros. Es por ese motivo por lo que no hemos creado un modelo. Después de creada, añadiremos los campos necesarios, en este caso las claves primarias de las dos tablas que estamos relacionando.
3. Hacemos un migrate para crear las tablas, y añadimos datos.
4. Creamos una función en el modelo User.php, para poder obtener todos los roles del usuario. Llamaremos a esta función roles, y será gestionada como un atributo que contenga todos los roles del usuario en un array de objetos.
5. Creamos la ruta en web.php para acceder a los roles de un usuario

1 y 2 .- Creación de modelos y migraciones

Vamos a crear el modelo de la tabla de roles con artisan, creando a la vez la migración (con -m)::

```
$ php artisan make:model Role -m
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:model Role -m

  INFO  Model [app/Models/Role.php] created successfully.

  INFO  Migration [database/migrations/2023_01_04_200655_create_roles_table.php] created successfully.
```

En la migración añadimos un campo para completar el diseño de la tabla:

2023_01_04_200655_create_roles_table.php

```
public function up()
{
    Schema::create('roles', function (Blueprint $table) {
        $table->id();
        $table->string('nombre');
        $table->timestamps();
    });
}
```

Ahora crearemos la tabla de relación. Laravel, por convención, nombra el modelo de relación en singular, role_user en este caso. Así, Laravel sabrá que es una tabla de relación.

```
$ php artisan make:migration create_users_roles_table --create=role_user
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:migration create_users_roles_table --create=ro
le_user

  INFO  Migration [2023_01_04_201503_create_users_roles_table] created successfully.
```

En la migración definimos los campos necesarios para la relación n a n

2023_01_04_201503_create_users_roles_table.php

```
public function up()
{
    Schema::create('role_user', function (Blueprint $table) {
        $table->id();
        $table->unsignedBigInteger('user_id');
        $table->unsignedBigInteger('role_id');
        $table->timestamps();
    });
}
```

Si seguimos la convención, y llamamos a las columnas de relación con el singular de sus tablas, Laravel manejará esta tabla de forma automática y no necesitaremos crear un modelo para programar el acceso a la misma.

3.- Migración y Alta de datos en las tablas

Migramos para crear las tablas en la base de datos:

```
$ php artisan migrate
```

Añadimos la siguiente información en las tablas:

users:

id	name	email	e
1	Marta Olmedilla	marta@eu.es	/
3	pepe	pepe@eu.es	/

roles:

id	nombre	created_at	updated_at
1	Administrador	2023-01-04 21:43:57	2023-01-04 21:43:57
2	Jefatura	2023-01-04 21:43:57	2023-01-04 21:43:57
3	Secretaría	2023-01-04 21:43:57	2023-01-04 21:43:57
4	Técnico	2023-01-04 21:43:57	2023-01-04 21:43:57

role_user:

id	user_id	role_id	created_at	updated_at
1	1	1	2023-01-04 21:47:07	2023-01-04 21:47:07
2	1	2	2023-01-04 21:47:07	2023-01-04 21:47:07
3	3	1	2023-01-04 21:47:42	2023-01-04 21:47:42
4	3	4	2023-01-04 21:47:42	2023-01-04 21:47:42

Vemos que el usuario 1 (marta) es tanto administrador (1) como jefe (2).
El usuario 3 (pepe), es tanto administrador (1) como técnico (4).

Nota: Para poner la fecha actual en un timestamp en MySQL puedes utilizar la función now().

4.- Creación de método “roles” en Users.php

Para obtener todos los roles de un usuario, utilizaremos una función roles, que podrá ser gestionada como variable de clase.

Users.php

```
    public function roles() {
        return $this->belongsToMany('App\Models\Role');
    }
```

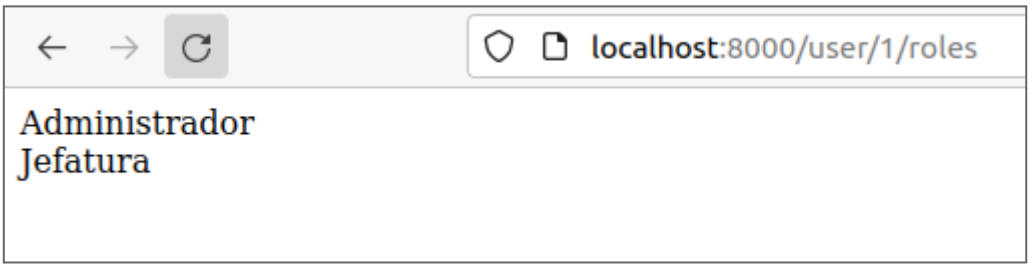
La función belongsToMany obtendrá todos los roles, su parámetro es el modelo Role.php, correspondiente a la tabla de roles.

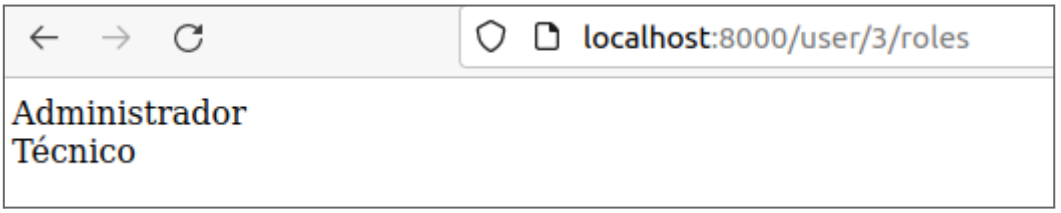
4.- Ruta para obtener roles de un usuario

web.php (opción 1)

```
// Relación muchos a muchos
Route::get('/user/{id}/roles', function($id) {
    $user = User::find($id);
    foreach($user->roles as $role) {
        echo $role->nombre . '<br>';
    }
})
```

Si accedemos, el resultado será el siguiente:





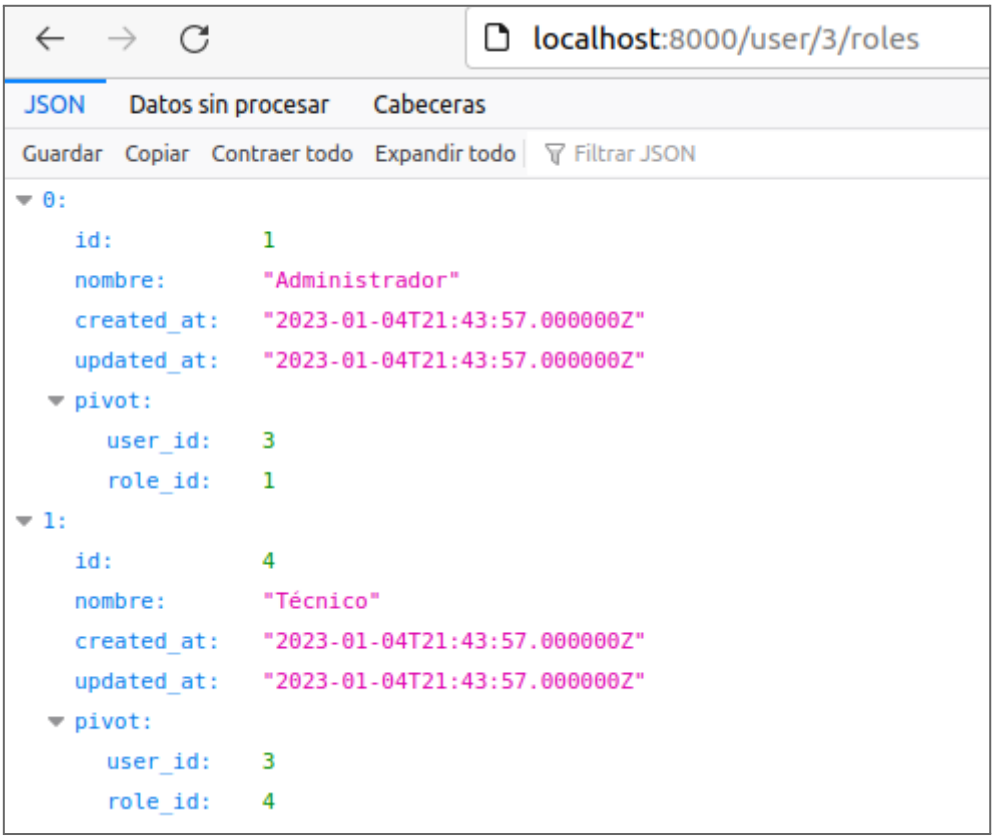
Como podemos ver, esta opción recupera los roles uno a uno, y los muestra con un echo, ya que el return sólo retornaría el primero.

web.php (opción 2)

```
// Relación muchos a muchos
Route::get('/user/{id}/roles', function($id) {
    $roles = User::find($id)->roles()->orderBy('id')->get();
    return $roles;
})
```

Esta opción recupera todos los roles en un objeto en formato JSON, indicando además un orden de salida. Aquí hacemos referencia a “roles” como método, no como atributo de clase.

Si accedemos a esta ruta, el resultado es el siguiente:



Al obtener la información, podemos ver que muestra

1. Los datos completos de cada rol del usuario 3 (pepe), en este caso el id del rol, el nombre y los timestamp
2. Los datos del registro de la tabla de relación que relaciona ese rol con el usuario indicado. “**pivot**” en inglés es una tabla de relación o tabla pivote.

Haciendo queries: Varios trucos

Eloquent ofrece muchos métodos con parámetros diversos, vamos a ver algo más de relaciones muchos a muchos.

Más parámetros en belongsToMany

En los métodos de Laravel podemos tener varios parámetros. Por ejemplo, en el método belongsToMany, un segundo parámetro podría ser el nombre de la tabla pivote, e incluso indicar un tercer y cuarto parámetros con los nombres de las foreign-key de la tabla pivote.

```
$this->belongsToMany('App\Models\Role', 'role_user', 'user_id',
'role_id);
```

En principio no es necesario informarlos, ya que hemos seguido la convención de nombres.

Acceso inverso: Usuarios de un post.

Para habilitarlo, crearemos un método en Role.php llamado users:

Role.php

```
    public function users() {
        return $this->belongsToMany('App\Models\User');
    }
```

Web.php

```
      // Usuarios de un rol concreto
      Route::get('/rol/{id}/users', function($id) {
          $users = Role::findOrFail($id)->users()
               ->orderBy('name')
               ->get();
          return $users;
      });
```

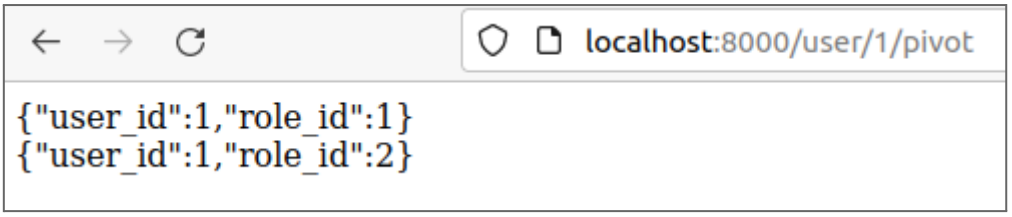
Acceso a la tabla intermedia de la relación (pivot)

Podemos acceder a la tabla intermedia, mediante el atributo pivot. Este atributo retornará un array de objetos, cada uno será un registro asociado de la tabla intermedia, en formato JSON. De dichos registros, sólo retornará los campos correspondientes a la relación (user_id y role_id):

web.php

```
Route::get('user/{id}/pivot', function ($id) {
    $user = User::find($id); // Array de objetos de User
    foreach($user->roles as $role) {
        echo $role->pivot . '<br>;
    }
})
```

El resultado es el siguiente

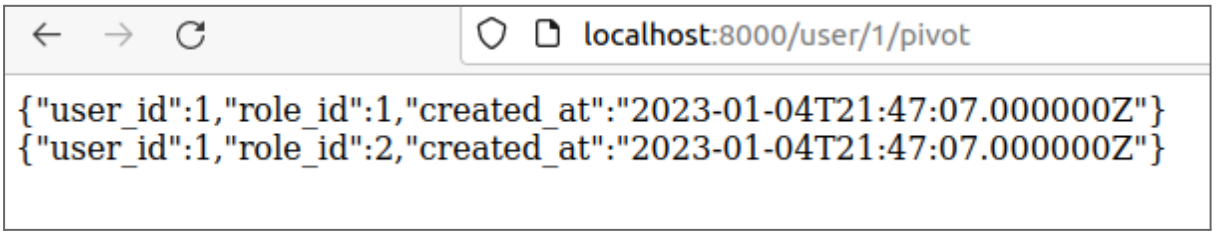


Si quisiera recuperar el resto de campos de la tabla pivote, en nuestro caso created_at, por ejemplo, tendríamos que modificar el modelo de User.php para que recupere esa información:

User.php

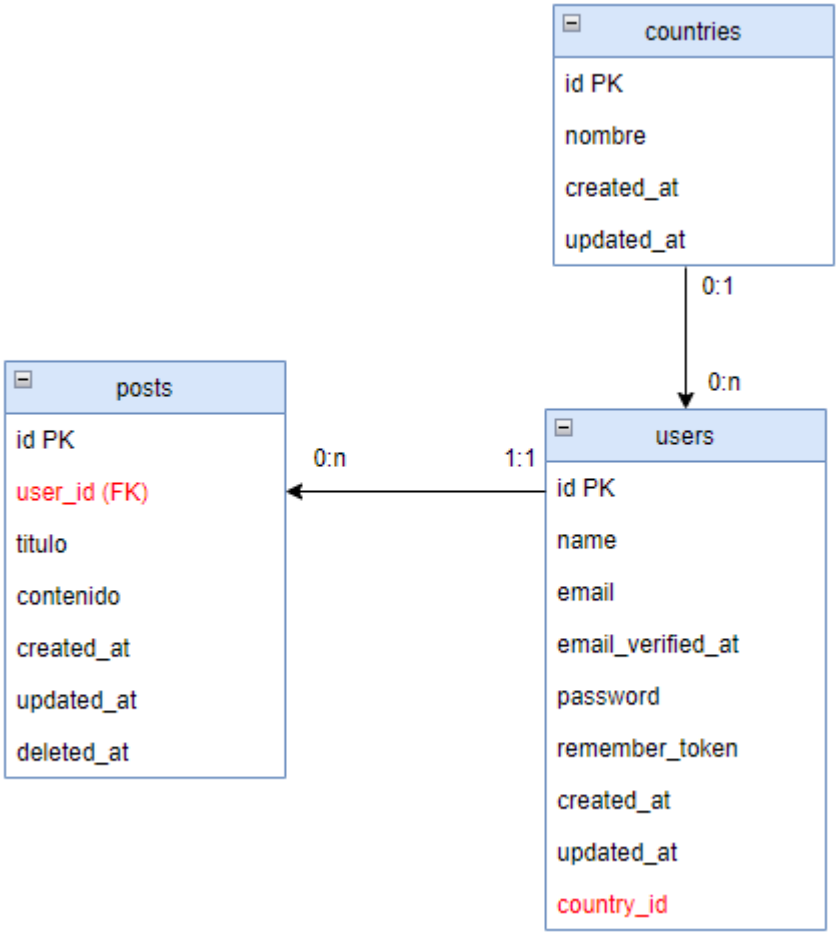
```
    public function roles() {
      return $this->belongsToMany('App\Models\Role')->withPivot('created_at');
    }
```

Si accedemos a la ruta anterior, obtendremos el siguiente resultado:



Relación has Many Through

Imaginemos que tenemos, además de la tabla usuarios y de la tabla posts, una tabla de países.



Ya vimos cómo acceder desde users a posts, con hasMany.

Si queremos, podemos configurar Laravel para recuperar, desde el modelo de paises, los posts de ese país. Laravel irá a través de la tabla users, de forma transparente a nosotros. Lo gestionará automáticamente.

1.- Creación del modelo y las migraciones

Para empezar, creamos los modelos y migraciones pendientes:

1. Creamos un modelo y una migración para la tabla de paises (countries):
   ```
   $ php artisan make:model Country -m
   ```
2. Añadimos pais_id a la tabla de usuarios (users), mediante una migración:
   ```
   $ php artisan make:migration add_country_id_column_to_users --table=users
   ```

El resultado es:

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:model Country -m

  INFO  Model [app/Models/Country.php] created successfully.

  INFO  Migration [database/migrations/2023_01_05_180831_create_countries_table.php] created successfully.
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:migration add_country_id_column_to_users --table=users

  INFO  Migration [2023_01_05_181137_add_country_id_column_to_users] created successfully.
```

Editamos la migración que añade la columna en la tabla users para incluirla en el up y eliminarla en el down:

2023_01_05_181137_add_country_id_column_to_users.php

```
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            //
            $table->unsignedBigInteger('country_id');
        });
    }
    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
            //
            $table->dropColumn('country_id');
        });
    }
```

A continuación editamos la migración que crea la tabla countries, para añadir una columna “nombre”.

2023_01_05_180831_create_countries_table.php

```
public function up()
    {
        Schema::create('countries', function (Blueprint $table) {
            $table->id();
            $table->string('nombre');
            $table->timestamps();
        });
    }
```

Finalmente, migramos:

```
$ php artisan migrate
```

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

  INFO  Running migrations.

  2023_01_05_180831_create_countries_table ..................................... 46ms DONE
  2023_01_05_181137_add_country_id_column_to_users ............................ 14ms DONE
```

Y añadimos información a la tabla de países y al campo nuevo añadido (country_id).

Como la migración de la tabla users sólo añade o quita una columna, los datos no se han borrado y no es necesario volver a incluirlos.

A continuación, añado al modelo de Country.php un método posts, que nos va a permitir obtener los posts de un país.

Country.php

```
    public function posts() {
        return
          $this->hasManyThrough('App\Models\Post','App\Models\User');
    }
```

La función **hasManyThrough** recupera todos los posts. Recibe como parámetro dos modelos, el primero es el de la tabla destino (posts), y el segundo es el de la tabla que atravesamos para llegar al destino (users).

Podríamos utilizar un tercer parámetro, indicando el nombre de la foreign key de la tabla intermedia (pais_id). Si quisiéramos personalizar el user_id de la tabla destino (posts), podríamos indicar su nombre como cuarto parámetro.

2.- Probamos creando datos y rutas

Añadimos la siguiente información:

Paises

id	nombre	created_at	updated_at
1	España	2023-01-05 19:27:18	2023-01-05 19:27:18
2	Canadá	2023-01-12 19:27:18	2023-01-05 19:27:18
3	Alemania	2023-01-05 19:27:46	2023-01-05 19:27:46
4	Méjico	2023-01-05 19:27:46	2023-01-05 19:27:46

Usuarios

id	name	email	email_verified_at	password	remember_token	created_at	updated_at	country_id
1	Marta Olmedilla	marta@eu.es	NULL		NULL	2023-01-04 21:46:00	2023-01-04 21:46:00	1
3	pepe	pepe@eu.es	NULL		NULL	2023-01-04 21:46:37	2023-01-04 21:46:37	4

Posts

id	user_id	titulo	contenido	created_at	updated_at	deleted_at
1	1	TITULO 1	CONTENIDO 1	2023-01-05 19:28:55	2023-01-05 19:28:55	NULL
2	1	TITULO 2	CONTENIDO 2	2023-01-05 19:32:02	2023-01-05 19:32:02	NULL
3	3	TITULO 3	CONTENIDO 3	2023-01-05 19:32:02	2023-01-05 19:32:02	NULL

Como puede verse, el usuario 1 (Marta) es de España(1) y tiene dos posts.
El usuario 3 (Pepe) es de Méjico (4) y tiene un post.

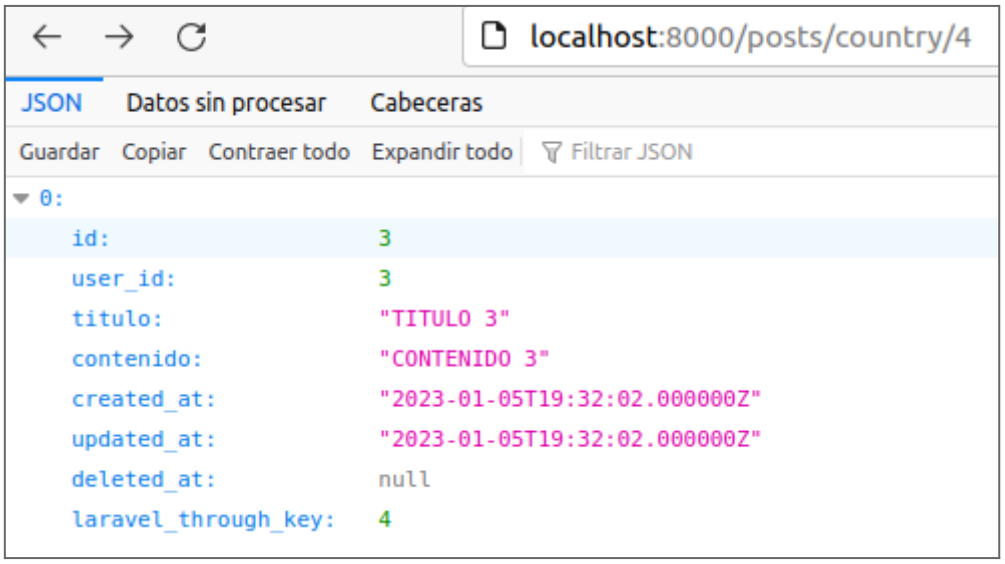
Si probamos la siguiente ruta, que contiene un parámetro id que es un código de país. Retornará cada uno de los posts de los usuarios de dicho país.

Web.php (Opción 1)

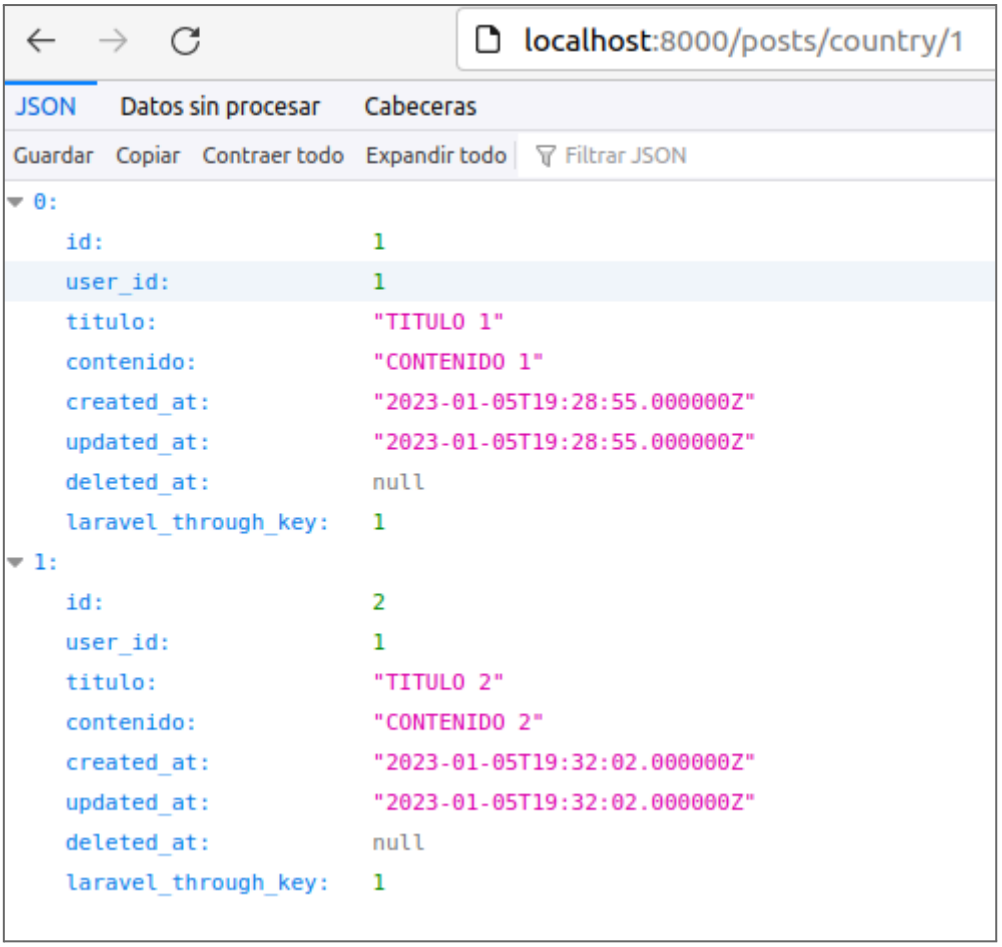
```
// Posts de un pais
use App\Models\Country;
Route::get('/posts/country/{id}', function ($id) {
    return Country::find($id)->posts;
});
```

En este caso, la función closure retorna en formato JSON los posts de un país. El resultado es como sigue:

<u>Posts del país 4 (Méjico)</u>



<u>Posts del país 1 (España):</u>



Laravel recupera la clave a través de la que ha recuperado los posts, en este caso el código del país. También recupera el user_id (tabla users), y el id del post (tabla posts)

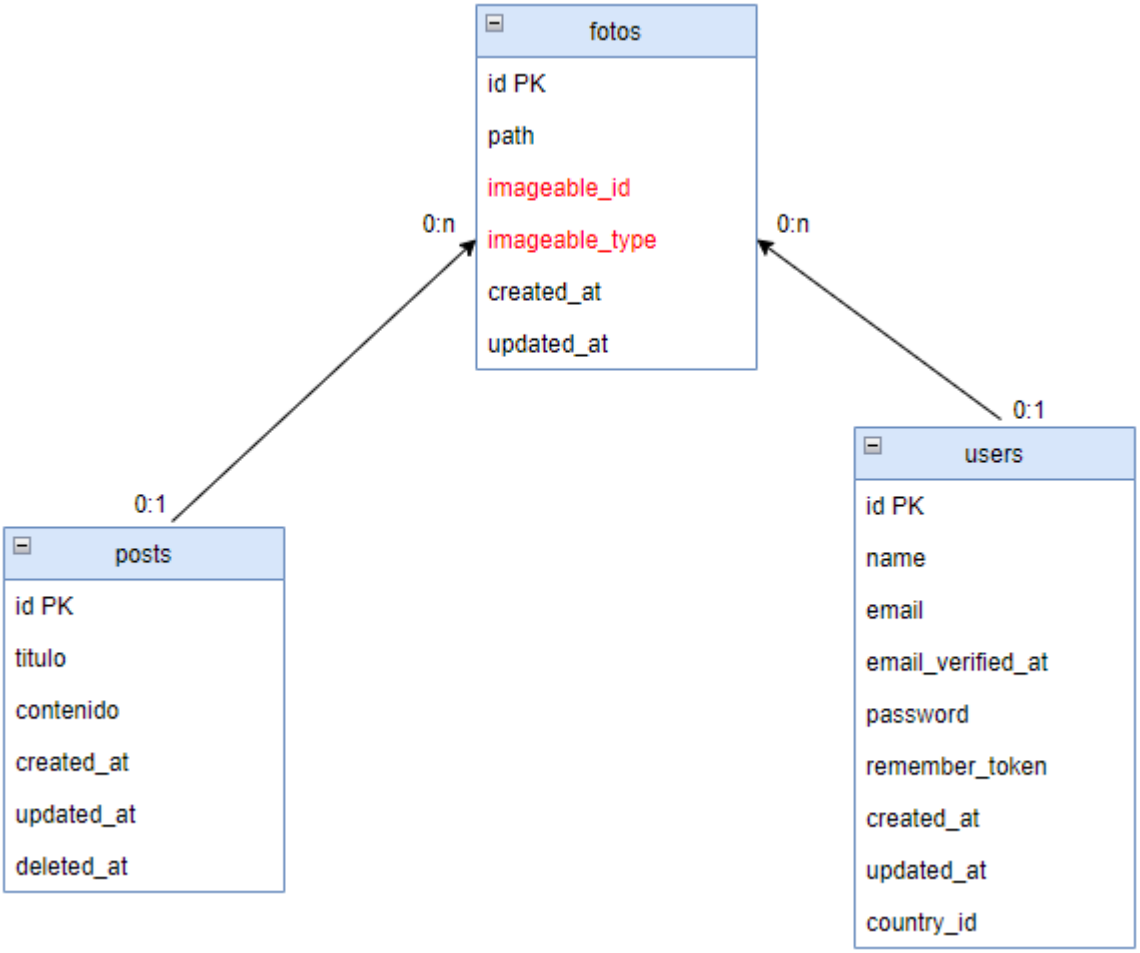
Web.php (Opción 2)

```
// Posts de un pais
use App\Models\Country;
Route::get('/posts/country/{id}', function ($id) {
    $pais = Country::find($id);
    foreach ($pais->posts as $post) {
        echo $post . '<br>';
    }
});
```

La ruta anterior permite formatear cada registro, uno a uno, de la forma que más nos guste.

Relación polimórfica

¿ Qué es una relación polimórfica ? Imaginemos que tenemos una tabla llamada fotos, y que tanto la tabla de usuarios como la tabla de posts están relacionadas con ella.



En esta relación, un usuario puede tener imágenes. Un post también podrá tener imágenes.

La tabla tendrá, por convención, dos columnas que implementan la relación:

- **imageable_id**: Es el id de la relación, la “foreign key”, que podrá ser de posts o de users.
- **imageable_type**: Indica si la relación es con posts o con users !!

Es habitual nombrar los campos con una palabra terminada en able. Por ejemplo, como tanto users como posts tienen imágenes, suele utilizarse la palabra imageable, que significa que ofrece imágenes para ser asociadas.

Este modelo es muy habitual, y muchos frameworks lo implementan, porque es flexible: Podemos añadir, por ejemplo, las fotos de un país, o las fotos de un rol, con indicar en el campo type qué tabla está relacionada, el id se buscaría en esa tabla. Además, la clase (el modelo) es única, y podrá ser utilizada por cualquier clase que pueda tener fotos asociadas, con un código muy limpio.

El inconveniente de este diseño, es que no permite implementar una foreign key en el modelo físico de la base de datos, con lo que podemos perder integridad. Toda la integridad se gestiona por programa. Por otro lado, las joins se complican, y si la tabla es muy grande se nota.

¿Vamos a utilizar relaciones polimórficas? En una base de datos relacional, no nos importa tener muchas tablas. En cambio, sí nos importa, y mucho, no poder implementar las foreign keys. El riesgo de inconsistencia es demasiado grande. En mi opinión, no deben utilizarse relaciones polimórficas en las bases de datos relacionales.

No obstante, en este apartado veremos cómo implementar una relación polimórfica en Laravel, ya que este tipo de relaciones se utilizan mucho.

1.- Creación de modelo y migración de la nueva tabla

Empezamos creando un modelo y una migración para la nueva tabla fotos.

```
$ php artisan make:model Foto -m
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:model Foto -m

  INFO  Model [app/Models/Foto.php] created successfully.

  INFO  Migration [database/migrations/2023_01_05_190630_create_fotos_table.php] created succes
sfully.
```

A continuación añadimos los campos que necesitemos a la migración y hacemos migrate para crear la tabla.

2023_01_05_190630_create_fotos_table.php

```php
public function up()
    {
        Schema::create('fotos', function (Blueprint $table) {
            $table->id();
            $table->string('path');
            $table->bigInteger('imageable_id');
            $table->string('imageable_type');
            $table->timestamps();
        });
    }
```

Hemos añadido varios campos, el path donde está la imagen, un id que será una clave foránea genérica, y un tipo que indicará de quién es foránea.

Los nombres imageable_id y imageable_type son una convención.

Finalmente crearemos la tabla:

```
$ php artisan migrate
```

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

  INFO  Running migrations.

  2023_01_05_190630_create_fotos_table ..............................................  59ms DONE
```

2.- Reestructuración de las relaciones

Eliminación de user_id de la tabla posts

Ahora tenemos una relación polimórfica, por lo que no nos interesa tener el user_id en la tabla posts:

- Quitamos el campo user_id de la migración que crea la tabla posts
- Hacemos php artisan migrate:refresh

Añadimos datos en la tabla fotos

Como hemos migrado todo, vuelvo a insertar registros.
En la tabla de fotos, el imageable_id podrá ser:

- Un id de usuario, en cuyo caso el tipo será el namespace de la clase User.php
- Un id de post, en cuyo caso el tipo será el namespace de la clase Post.php.

Insertamos los siguientes registros:

id	path	imageable_id	imageable_type	created_at	updated_at
1	foto1.jpg	1	App\Models\User	2023-01-05 20:27:52	2023-01-05 20:27:52
2	foto2.jpg	1	App\Models\Post	2023-01-05 20:33:02	2023-01-05 20:33:02

Como vemos, foto1.jpg pertenece al usuario 1, y foto2.jpg pertenece al post 1.

Vamos a escribir el código del modelo

En el modelo de Foto.php, crearemos una función llamada imageable (por convención coincidirá con el prefijo del nombre de los campos). Esta función utilizará la función morphTo.

Foto.php

```
    public function imageable() {
        return $this->morphTo();
    }
```

Para obtener las fotos de un post, tendremos que crear una función, dentro de Post.php, llamada fotos, que recuperará la información.

Post.php

```
    public function fotos() {
        return $this->morphMany('App\Models\Foto', 'imageable');
    }
```

La función **morphMany** recupera todos los registros de la tabla fotos. Tiene dos parámetros, el primero es el modelo de la tabla fotos, se referencia por su espacio de nombres, el segundo es el nombre del método que retorna el morphTo de una foto.

En usuarios tendremos que hacer lo mismo que en Posts.php, para que puedan recuperarse las fotos de un usuario:

Users.php

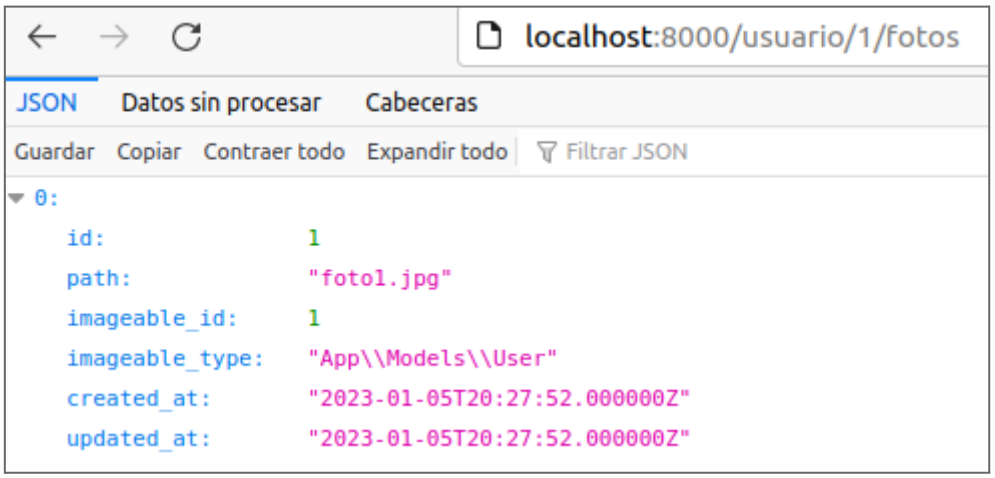
```
    public function fotos() {
        return $this->morphMany('App\Models\Foto', 'imageable');
    }
```

3.- Creación de rutas para probarlo

web.php (opción 1: Fotos de un usuario)

```
// Relación polimórfica
Route::get('/usuario/{id}/fotos', function($id) {
    return User::find($id)->fotos;
})
```

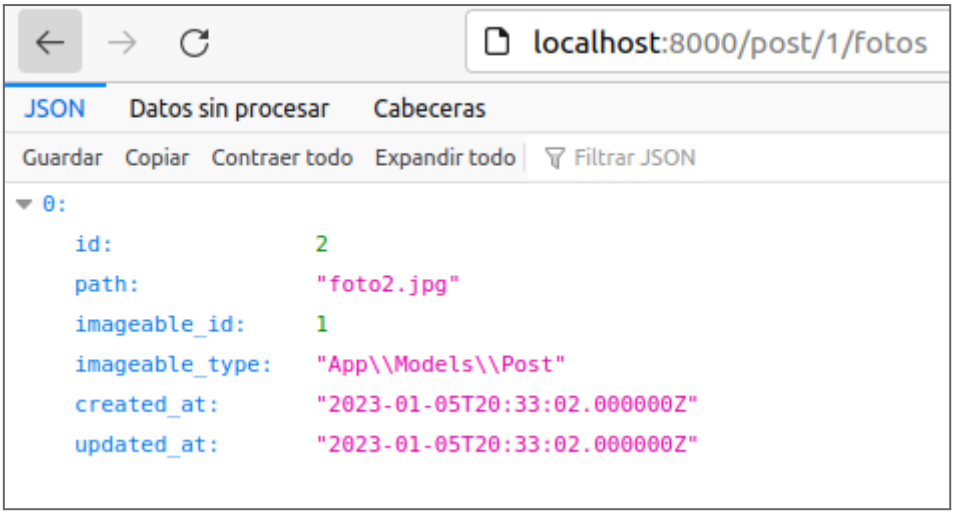
Si accedemos a esta ruta con el usuario 1, obtendremos su foto:



web.php (opción 2: Fotos de un post)

```
// Fotos de un post
Route::get('/post/{id}/fotos', function($id) {
    return Post::find($id)->fotos;
});
```

Si accedemos a esta ruta con el post 1, obtendremos su foto:



Por supuesto, también podríamos programar una ruta que nos permita recorrer las fotos una a una para formatear con echo el resultado, en vez de hacer un return de toda la información en formato JSON, que es lo que hemos mostrado aquí.

4.- Recuperación inversa

Si lo que queremos es recuperar el propietario de una foto, que podría ser un usuario o un post, lo haríamos de la siguiente forma:

web.php

```
// Propietario de una foto
use App\Models\Foto;
Route::get('/foto/{id}/owner', function($id) {
    return Foto::findOrFail($id)->imageable;
})
```

Aquí he utilizado **findOrFail**, en vez de find. Con esta función, si no existe la foto se retorna "404 NOTFOUND", en vez de un error feo.

En cuanto a la función imageable, lo que retorna es la información del propietario, sea quien sea:

- De la columna imageable_type obtiene el espacio de nombres del modelo con quien se relaciona esa foto
- Accede con el imageable_id a la tabla indicada anteriormente
- Recupera el registro de dicha tabla.

Por supuesto, estamos en relación polimórfica 1 a n.

El resultado de este código, para ambas fotos, sería:

<u>Foto 1: El propietario es un usuario:</u>

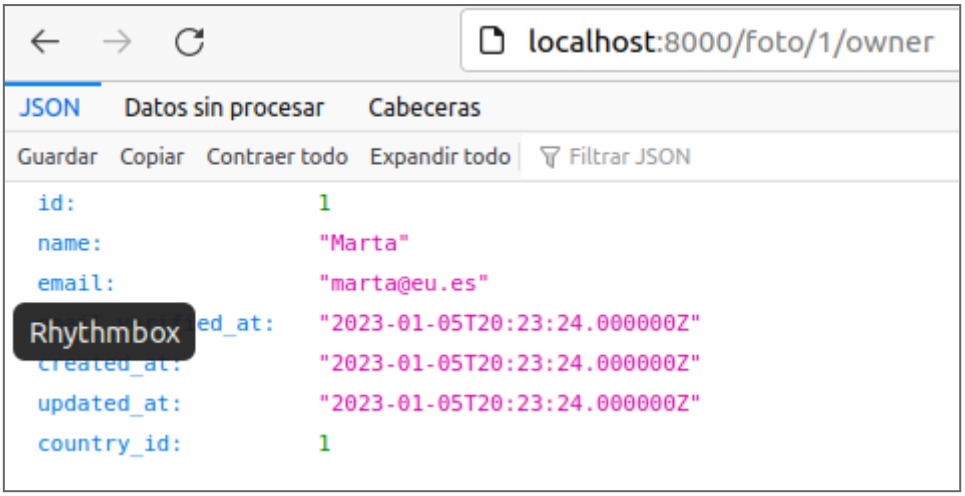
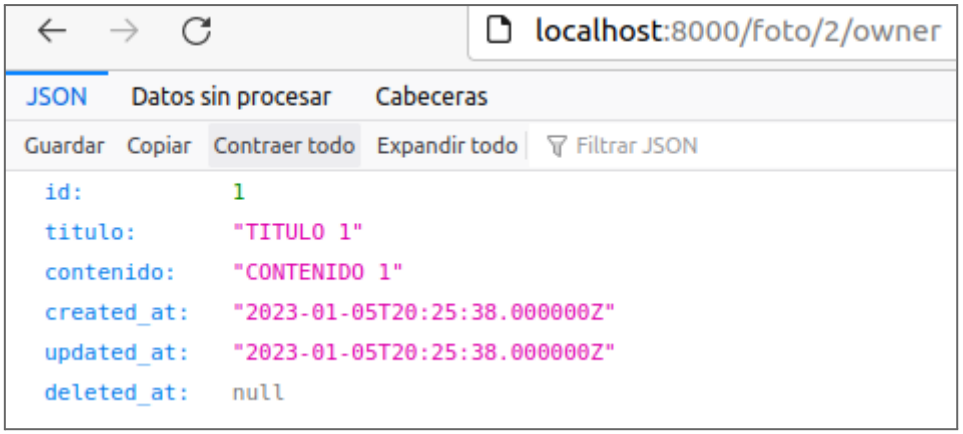


Foto 2: El propietario es un post



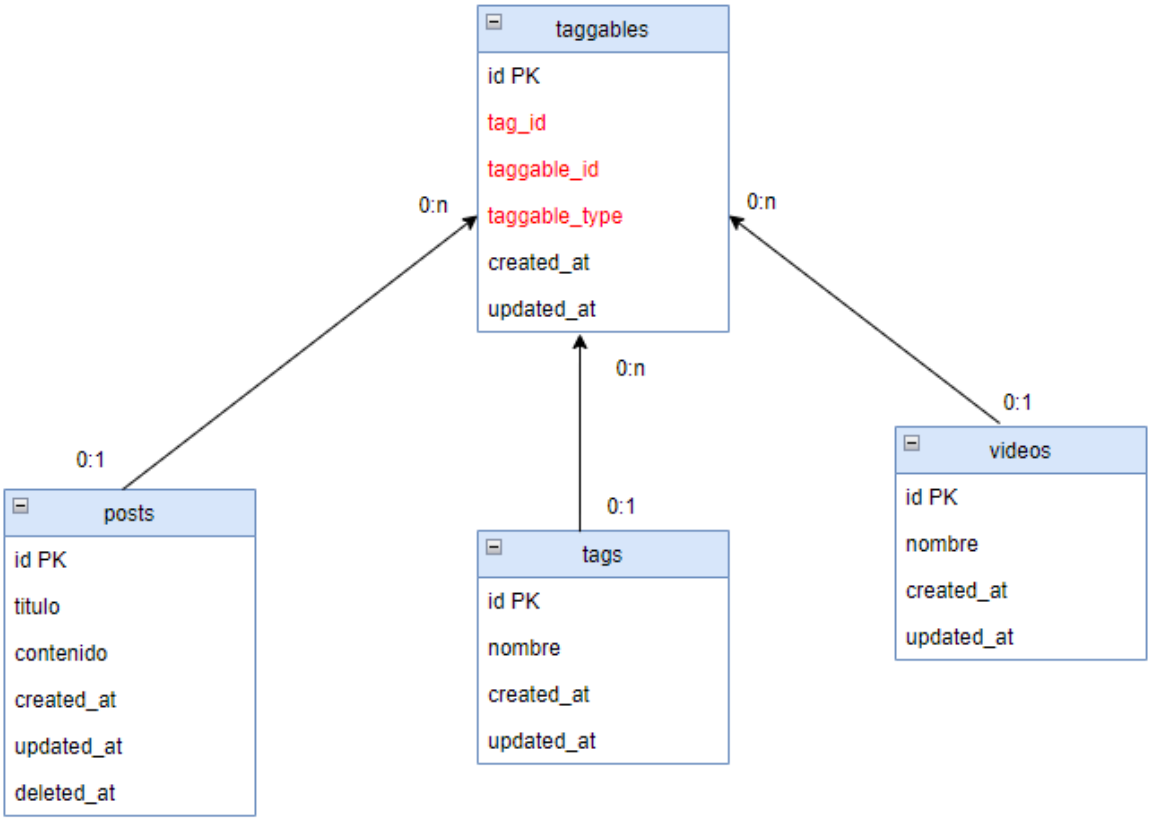
Como puede verse, Laravel recupera el propietario de la tabla que corresponde de forma automática.

5.- Relación polimórfica n a n

Imaginemos que queremos asignar etiquetas a nuestros vídeos, posts, etc. Un vídeo podrá tener varias etiquetas, y una etiqueta podrá pertenecer a varios vídeos, tal y como es una relación n a n típica. Por otro lado, no sólo los vídeos son etiquetables, también pueden serlo los posts, etc., lo que nos lleva a una relación polimórfica.

Necesitaremos, para normalizar la relación muchos a muchos, una tabla de relación intermedia. La llamaremos, por convención “etiquetables”, en inglés “taggables” (tag en inglés es etiqueta).

Un modelo de datos que refleje esta situación podría ser el siguiente:



Creación

Creamos un modelo y una migración para dos tablas nuevas: videos, tags y taggables (etiquetables)

$ php artisan make:model Video -m
$ php artisan make:model Tag -m
$ php artisan make:model Taggable -m

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:model Video -m

  INFO  Model [app/Models/Video.php] created successfully.

  INFO  Migration [database/migrations/2023_01_10_091726_create_videos_table.php] created successfully.
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:model Tag -m

  INFO  Model [app/Models/Tag.php] created successfully.

  INFO  Migration [database/migrations/2023_01_10_091854_create_tags_table.php] created successfully.
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacion1$ php artisan make:Model Taggable -m

  INFO  Model [app/Models/Taggable.php] created successfully.

  INFO  Migration [database/migrations/2023_01_10_092101_create_taggables_table.php] created successfully.
```

Migraciones

A continuación modificamos las migraciones generadas para completar los campos:

En la creación de videos y tags, añadimos un atributo “nombre” de tipo string

2023_01_10_091726_create_videos_table.php

```
public function up()
    {
        Schema::create('videos', function (Blueprint $table) {
            $table->id();
            $table->string('nombre');
            $table->timestamps();
        });
    }
```

2023_01_10_091854_create_tags_table.php

```
public function up()
    {
        Schema::create('tags', function (Blueprint $table) {
            $table->id();
            $table->string('nombre');
            $table->timestamps();
        });
    }
```

2023_01_10_092101_create_taggables_table.php

```
    public function up()
    {
        Schema::create('taggables', function (Blueprint $table) {
               // Relación con tabla tag
            $table->bigInteger('tag_id');
               // Id de elemento etiquetado
            $table->bigInteger('taggable_id');
               // TIpo de elemento etiquetado
            $table->string('taggable_type');
        });
    }
```

Modelos

Después modificamos los modelos
En Posts, queremos poder obtener las etiquetas de un post concreto:

Post.php

```
    public function tags() {
        return $this->morphToMany('App\Models\Tag', 'taggable');
    }
```

En tags, querremos poder obtener todos los posts o todos los videos que tengan una etiqueta concreta

Tag.php

```
    public function posts() {
        return $this->morphedByMany('App\Models\Post','taggable');
    }

    public function videos() {
        return $this->morphedByMany('App\Models\Video','taggable');
    }
```

Recuperando información

Primero vamos a meter datos en las tablas, para ello las creamos mediante artisan migrate.

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

  INFO  Running migrations.

  2023_01_10_091726_create_videos_table ..................................... 65ms DONE
  2023_01_10_091854_create_tags_table ....................................... 51ms DONE
  2023_01_10_092101_create_taggables_table .................................. 43ms DONE
```

Añadimos a las tablas videos, tags y taggables información desde PHPMyAdmin:

- Relacionamos varias etiquetas para algún vídeo
- Relacionamos varias etiquetas para algún post

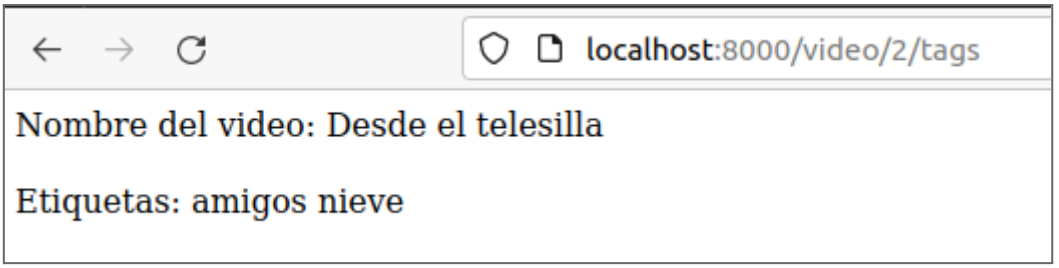
Finalmente, creamos algunas rutas en web.php para recuperar información:

web.php

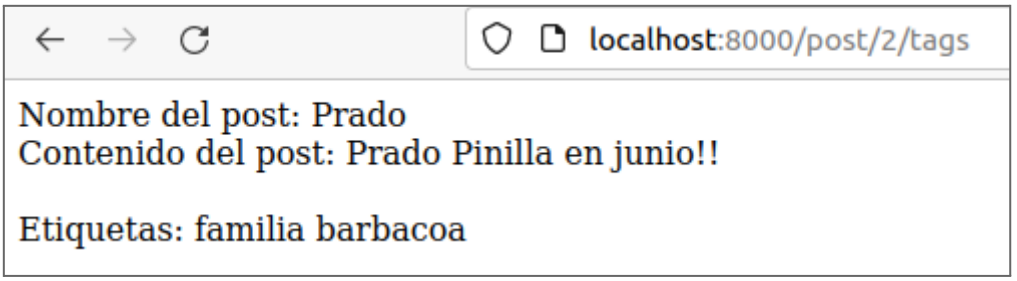
```
// Etiquetas de un post
Route::get('/post/{id}/tags', function($id){
    $post = Post::find($id);
    echo "Nombre del post: " . $post->titulo . "<br>";
    echo "Contenido del post: " .$post->contenido ."<br><br>Etiquetas: ";
    foreach($post->tags as $tag) {
        echo $tag->nombre . " ";
    }
});

// Etiquetas de un video
use App\Models\Video;
Route::get('/video/{id}/tags', function($id){
    $video = Video::find($id);
    echo "Nombre del video: " . $video->nombre . "<br><br>Etiquetas: ";
    foreach($video->tags as $tag) {
        echo $tag->nombre . " ";
    }
});
```

Si abrimos las rutas en un navegador, el resultado será:

Tags de un video:



Tags de un post:



Recuperación inversa:

Si queremos recuperar los videos, o los posts, correspondientes a una etiqueta concreta, podemos hacer las siguientes rutas:

web.php

```
// Posts con un tag concreto
use App\Models\Tag;
Route::get('/tag/{id}/posts', function($id){
    $tag = Tag::find($id);
    echo "Posts de la etiqueta: " . $tag->nombre . "<br><br>";
    foreach ($tag->posts as $post) {
        echo $post->titulo . "<br>";
    }
});

// Videos con un tag concreto
Route::get('/tag/{id}/videos', function($id){
    $tag = Tag::find($id);
    echo "Videos de la etiqueta: " . $tag->nombre . "<br><br>";
    foreach ($tag->videos as $video) {
        echo $video->nombre . "<br>";
    }
});
```

Si llamamos a estas rutas desde el navegador:

Videos de un tag

Posts de un tag