# Create New Project & Database

$ laravel new project\_name => 建立一個新的laravel project

建立好後，到.env中設定連結的資料庫名稱與登入的帳號密碼，設定完後執行migrate

$ php artisan migrate => 會將預設的遷移檔案搬運到資料庫中

$ php artisan make:model Address –m => 新增一個Address的模型

    public function up()

    {

        Schema::create('addresses', function (Blueprint $table) {

            $table->id();

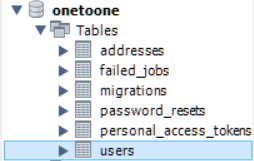
            $table->string('name');

            $table->timestamps();

        });

    }

到遷移檔案設置需要的欄位



但是需要使用one to one relationship，因此需要在addresses中新增user\_id欄位

    public function up()

    {

        Schema::create('addresses', function (Blueprint $table) {

            $table->id();

            $table->integer('user\_id')->unsigned()->nullable();

            $table->string('name');

            $table->timestamps();

        });

    }

將user\_id設置為正數且可為空(Null)

$ php artisan migrate:refresh => 因為初期沒資料，可以用refresh的方式將新增的欄位重新填入資料庫中



# One to One

**Create One to One Data**

use App\Models\Address;

use App\Models\User;

Route::get('/insert', function() {

    $user = User::findOrFail(1);

    $address = new Address(['name'=>'Taipei']);

    $user->address()->save($address);

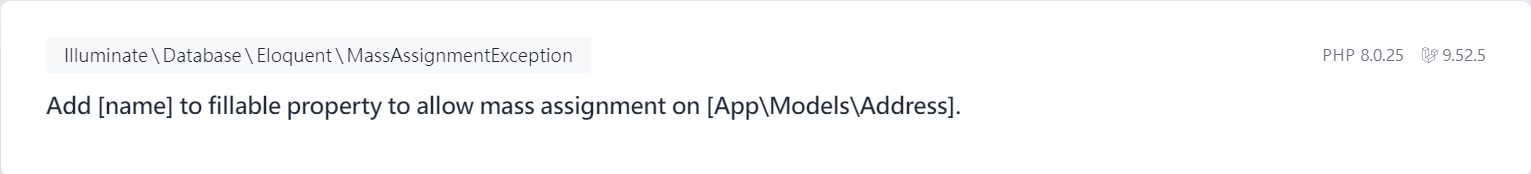
});

先將需要使用到的Model引用進web.php中

$user儲存在資料庫中id=1的資料

$address插入一筆新的資料name=Taipei

$user->address()->save($address); => $user是先前搜尋到id=1的user資料，透過Model中的hasOne()與Address建立one to one的關係，並且儲存該筆資料



到localhost:8000/insert頁面後出現上述的錯誤資訊，需要在Address中新增fillable => 可以大量修改資料的欄位

class Address extends Model

{

    use HasFactory;

    protected $fillable = [ 'name' ];

}

指定哪些可以存(fillable)

設置完畢後，再次到localhost:8000/insert頁面，會顯示空白頁面，再到資料庫中進行檢查



可以看到name=Taipei的資料被輸入到addresses中，並且沒有設置的user\_id也同樣顯示出來

**Update One to One Data**

Route::get('/update', function() {

    $address = Address::where('user\_id', '1');

    $address = Address::whereUserId(1)->first();

});

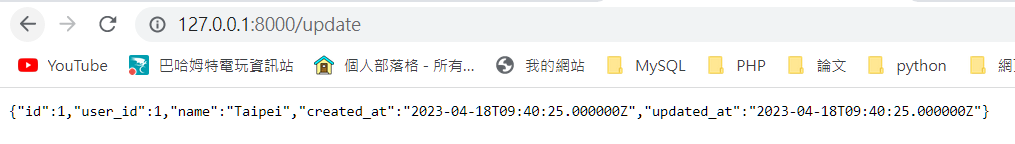
有不同種where條件式列法可以使用

Route::get('/update', function() {

    $address = Address::whereUserId(1)->first();

    return $address;

});



Route::get('/update', function() {

    $address = Address::whereUserId(1)->first();

    $address->name = "New Taipei City";

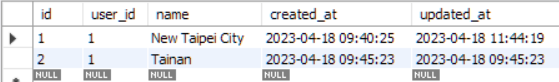
    $address->save();

});

將addresses中id=1的第一筆資料獲取並賦值給$address

$address中的name欄位重新命名

再將$address儲存到資料庫中



可以看到第一筆資料的name欄位已經被順利更新

**Delete One to One Data**

Route::get('/delete', function() {

    $user = User::findOrFail(1);

    $user->address()->delete();

    return "Delete complete!";

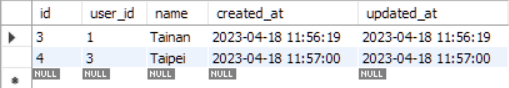
});



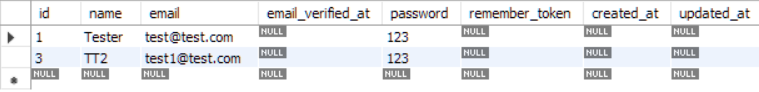


由於user\_id=1綁定兩個address的資料，因此刪除時，是將兩筆資料一起刪除

重新測試一次



Address新資料，分別綁定不同user



user\_1和user\_3

Route::get('/delete', function() {

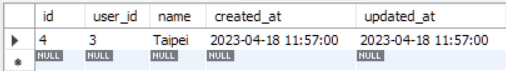
    $user = User::findOrFail(1);

    $user->address()->delete();

    return "Delete complete!";

});

同樣刪除user\_id=1的address資料



成功刪除資料，只刪除user\_id=1的資料

# One to Many

**Setting one to many project**

$ laravel new onetomany => 新建立一個onetomany的專案

$ php artisan make:model Post –m => 建立Post的Model => 一個user可以有多個posts

    public function up()

    {

        Schema::create('posts', function (Blueprint $table) {

            $table->id();

            $table->integer('user\_id')->unsigned()->nullable()->index();

            $table->string('title');

            $table->text('body');

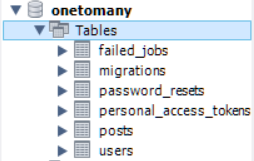
            $table->timestamps();

        });

    }

在Post模型中設置以上欄位

$ php artisan migrate => 設置完畢後，同步到資料庫中



**Create One to Many Data**

    public function post() {

        return $this->hasMany('App\Models\Post');

    }

先到User的模型中設置與Post的關聯，一個user可以有多個posts，因此使用hasMany()

class Post extends Model

{

    use HasFactory;

    protected $fillable = [

        'title',

        'body'

    ];

}

將title和body的欄位權限打開，才能在後續進行插入/修改/刪除資料

Route::get('/insert', function() {

    $user = User::findOrFail(1);

    $post = new Post(['title'=>'Second post', 'body'=>'This is the second one to many post']);

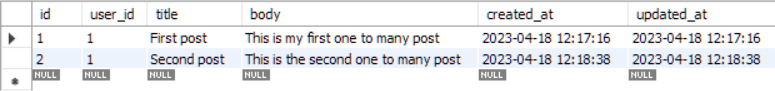
    $user->post()->save($post);

});

先查詢已存在的user

再建立一個新的post

建立user與post的關聯，並且將post存進資料庫中



**Read One to Many Data**

Route::get('/read', function() {

    $user = User::findOrFail(1);

    return dd($user->post);

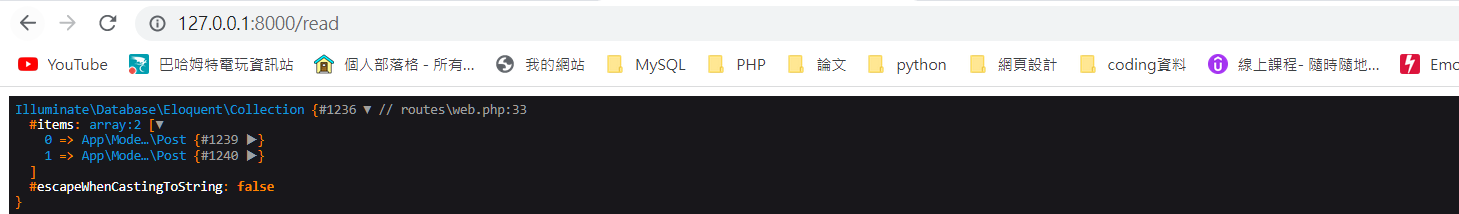
    // foreach($user->post as $post) {

    //     echo "title : ".$post->title."<br> content: ".$post->body."<br>";

    // }

});

dd() 是開發過程中最常用的函式之一，可以印出送入的參數，並且暫停程式。



Route::get('/read', function() {

    $user = User::findOrFail(1);

    foreach($user->post as $post) {

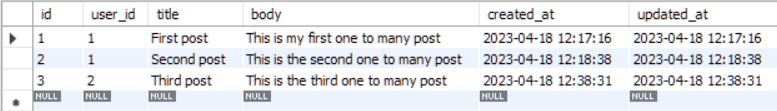
        echo "title : ".$post->title."<br> content: ".$post->body."<br>";

    }

});

將屬於user\_id=1的post給列印出來





**Update One to Many Data**

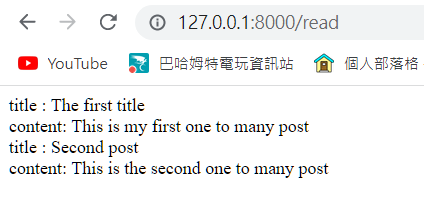
Route::get('/update', function() {

    $user = User::findOrFail(1);

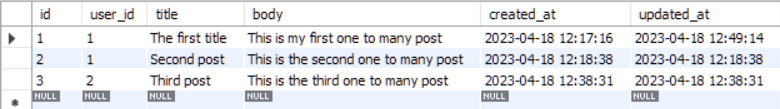
    $user->post()->whereId(1)->update(['title'=>'The first title']);

});

透過user去尋找post，並且修改post\_id=1的title



修改完後可以看到First post變成The first title



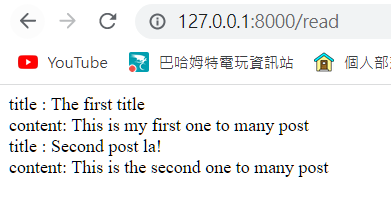
Route::get('/update', function() {

    $user = User::findOrFail(1);

    $user->post()->where('id', '=', 2)->update(['title'=>'Second post la!']);

});

使用另一種條件式去修改post的資料



**Delete One to Many Data**

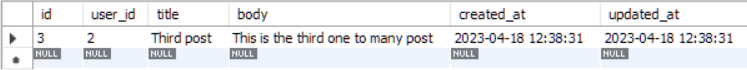
Route::get('/delete', function() {

    $user = User::findOrFail(1);

    $user->post()->delete();

});

透過user去查找user\_id=1底下的所有post文章，並且全部刪除



可以看到原先的post\_id=1, post\_id=2的資料都被刪除了

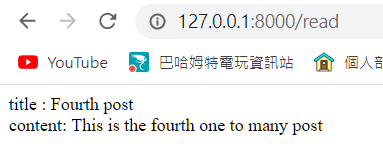
Route::get('/delete', function() {

    $user = User::findOrFail(2);

    $user->post()->whereId(3)->delete();

});

或是使用條件式刪除特定的post



第三篇post已經被刪除

# Many to Many

**Setting many to many project**

laravel new manytomany => 建立manytomany的laravel專案

在.env中setting資料庫相關資訊

$ php artisan make:model Role –m => 建立Role的模型

$ php artisan make:migration create\_role\_user\_table --create=role\_user => 建立user和role的中介表

    public function up()

    {

        Schema::create('role\_user', function (Blueprint $table) {

            $table->id();

            $table->integer('user\_id')->unsigned()->nullable()->index();

            $table->integer('role\_id')->unsigned()->nullable()->index();

            $table->timestamps();

        });

    }

在role\_user的遷移檔案中設置中介的表格id，user\_id和role\_id

    public function up()

    {

        Schema::create('roles', function (Blueprint $table) {

            $table->id();

            $table->string('name');

            $table->timestamps();

        });

    }

在role遷移檔案中設置需要新增的欄位

以上設置完畢後，將資料同步到資料庫中

$ php artisan migrate

**Create Many to Many Data**

    public function roles() {

        return $this->belongsToMany('App\Models\Role');

    }

先到User的模型中設置與Role模型的關係

class Role extends Model

{

    use HasFactory;

    protected $fillable = [  'name'  ];

}

再到Role中新增可以修改的欄位 ‘name’

Route::get('/insert', function() {

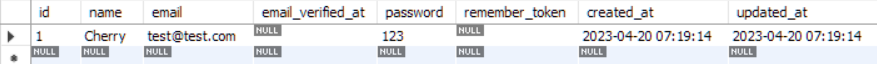
    $user = User::find(1);

    $role = new Role(['name'=>'administrator']);

    $user->roles()->save($role);

});

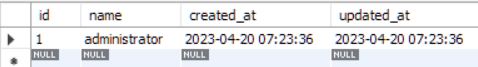
回到web.php中設置需要插入的資料 => 需要先插入一筆users的資料

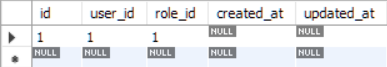


$user儲存這筆資料

$role創建新的一筆資料，name=administrator

$user透過Model中的roles()，執行多對多關係，將roles產生的新資料和users的id資料，傳入進role\_user的中介表中





**Read Many to Many Data**

Route::get('/read', function() {

    $user = User::findOrFail(1);

    foreach($user->roles as $role) {

        dd($role);

    }

});

透過user尋找該user擁有的role資料



Route::get('/read', function() {

    $user = User::findOrFail(1);

    return dd($user->roles);

    // foreach($user->roles as $role) {

    //     dd($role);

    // }

});

會獲得一個collection



Route::get('/read', function() {

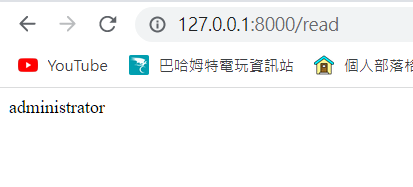
    $user = User::findOrFail(1);

    foreach($user->roles as $role) {

        return $role->name;

    }

});



**Update Many to Many Data**

Route::get('/update', function() {

    $user = User::find(1);

    foreach($user->roles as $role) {

        $role->name = 'subscriber';

        $role->save();

    }

});

透過user去搜尋role，再修改role的資料

Route::get('/update', function() {

    $user = User::find(1);

    if($user->has('roles')) {

        foreach($user->roles as $role) {

            if($role->name != 'administrator') {

                $role->name = 'administrator';

                $role->save();

                return $role->name;

            } else {

                $role->name = 'subscriber';

                $role->save();

                return $role->name;

            }

        }

    }

});

使用if-else更新資料

**Delete Many to Many Data**

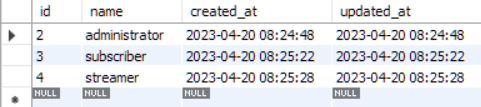
Route::get('/delete', function() {

    $user = User::findOrFail(1);

    $user->roles()->delete();

});

此方式會將roles中的所有資料給刪除



先添加其他資料回roles中

Route::get('/delete', function() {

    $user = User::findOrFail(1);

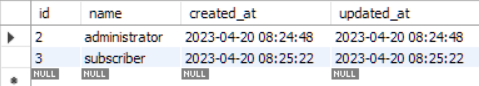
    foreach($user->roles as $role) {

        $role->whereId(4)->delete();

    }

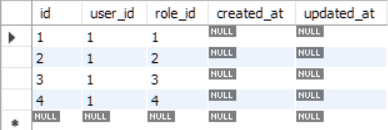
});

使用where條件式篩選需要刪除的資料，將id=4的資料給刪除



**Attaching, detaching and syncing**

* Attaching



role\_user的中介表目前有這些資料

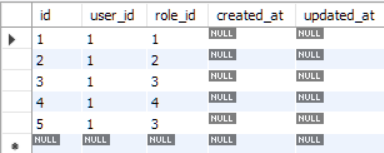
Route::get('/attach', function() {

    $user = User::findOrFail(1);

    $user->roles()->attach(3);

});

先搜尋使用者id=1的資料，並且將該資料與roles中id=3的資料進行many to many的連結



可以看到user\_id=1和role\_id=3的資料被新增進中介表中

* Detaching

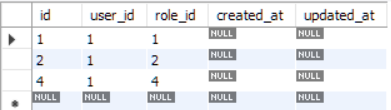
Route::get('/detach', function() {

    $user = User::findOrFail(1);

    $user->roles()->detach(3);

});

與attach有相同概念，detach為斷開user與role的關聯



執行後會發現user\_id=1和role\_id=3的資料都被刪除掉了

    $user->roles()->detach();

若沒指定數值，則會將user與role的所有連結給刪除



* Sync

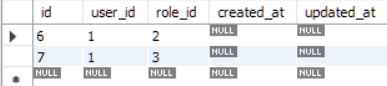
Route::get('/sync', function() {

    $user = User::findOrFail(1);

    $user->roles()->sync([2,3]);

});

與attach類似，會將user\_id=1和role\_id=2,3的資料進行連結



role\_user的表格 (原先已經被detach給刪除到沒有資料)

#注意

sync()內部傳入的變數為array，並且只與array中的資料進行連結，其他資料將被刪除

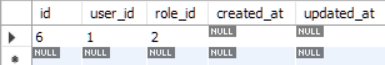
Route::get('/sync', function() {

    $user = User::findOrFail(1);

    $user->roles()->sync([2]);

});

例如這邊的user\_id=1與role\_id=2的資料進行連結，先前user\_id=1與role\_id=3的資料將被刪除



# Polymorphic Relationship

**Setting polymorphic project**

laravel new polymorphic => 建立polymorphic的laravel專案

$ php artisan make:model Staff –m => 建立staffs表格

$ php artisan make:model Product –m => 建立產品表格

$ php artisan make:model Photo –m => 建立相片表格

    public function up()

    {

        Schema::create('staff', function (Blueprint $table) {

            $table->id();

            $table->string('name');

            $table->timestamps();

        });

    }

到staff的遷移檔案設置名稱欄位

    public function up()

    {

        Schema::create('products', function (Blueprint $table) {

            $table->id();

            $table->string('name');

            $table->timestamps();

        });

    }

到product的遷移檔案設置名稱欄位

    public function up()

    {

        Schema::create('photo', function (Blueprint $table) {

            $table->id();

            $table->string('path');

            $table->integer('imageable\_id');

            $table->string('imageable\_type');

            $table->timestamps();

        });

    }

到photo的遷移檔案設置路徑欄位和多型用的id和type欄位

class Photo extends Model

{

    use HasFactory;

    protected $fillable = [ 'path' ];

    public function imageable() {

        return $this->morphTo();

    }

}

到Photo的Model設置可以修改的欄位 = path，並且建立多型的關係

class Staff extends Model

{

    use HasFactory;

    protected $fillable = [ 'name' ];

    public function photos() {

        return $this->morphMany('App\Models\Photo', 'imageable');

    }

}

到Staff的Model設置可修改欄位 name，並且建立與Photo之間的關係，調用Photo的imageable()

class Product extends Model

{

    use HasFactory;

    protected $fillable = [ 'name' ];

    public function photos() {

        return $this->morphMany('App\Models\Photo', 'imageable');

    }

}

到Product的Model設置可修改欄位 name，並且建立與Photo之間的關係，調用Photo的imageable()

**Create Polymorphic Data**

Route::get('/insert', function() {

    $staff = Staff::findOrFail(1);

    $staff->photos()->create(['path'=>'example.jpg']);

});

透過staff建立與photos的關係



photos有一張exmaple.jpg的圖片路徑；並且是與staff\_id=1和Staff Model有關連

**Reading Polymorphic Data**

Route::get('/read', function() {

    $staff = Staff::findOrFail(1);

    foreach($staff->photos as $photo) {

        return $photo->path;

    }

});

先前有建立staff和photo的關聯

因此透過staff去尋找photo的path資料

Route::get('/read', function() {

    $photo = Photo::find(1);

    return $photo->imageable->name;

});

反向搜尋，透過photo搜尋staff資料

**Update Polymorphic Data**

Route::get('/update', function() {

    $staff = Staff::findOrFail(1);

    $photo = $staff->photos()->whereId(1)->first();

    $photo->path = 'change.jpg';

    $photo->save();

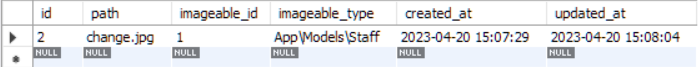
});

先搜尋staff\_id=1的資料

透過staff去搜尋photo的資料，並且搜尋id=1的photo

設定photo的path

再同步到資料庫中



**Delete Polymorphic Data**

Route::get('/delete', function() {

    $staff = Staff::findOrFail(1);

    $photo = $staff->photos()->whereId(1)->first();

    return $photo->delete();

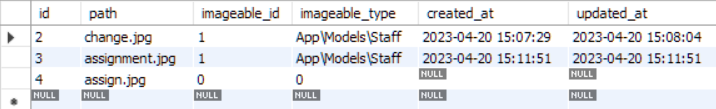
});

先搜尋staff

再透過staff搜尋圖片並且刪除他



**Extra techniques**



先插入一筆資料，imageable\_id和imageable\_type為隨便的值

Route::get('/assign', function() {

    $staff = Staff::findOrFail(1);

    $photo = Photo::findOrFail(4);

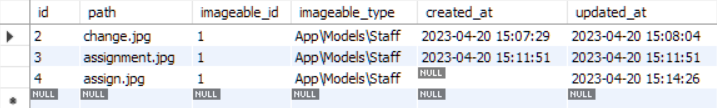
    $staff->photos()->save($photo);

});

先搜尋staff\_id=1的資料

再搜尋先前插入imageable\_id和imageable\_type為0的資料

將其連結的對象為staff\_id=1並且同步更新到資料庫中



Route::get('/unassign', function() {

    $staff = Staff::findOrFail(1);

    $staff->photos()->whereId(4)->update(['imageable\_id'=>0, 'imageable\_type'=>0, 'updated\_at'=>Null]);

});

先前是將staff和photo進行連結，此處則是斷除photo和staff的連結

# Polymorphic Many to Many Relationship

**Setting polymorphic many to many project**

$ laravel new polymorphicmanytomany => 建立polymorphic的laravel專案

$ php artisan make:model Post –m => 建立posts表格

$ php artisan make:model Video –m => 建立videos表格

$ php artisan make:model Tag –m => 建立tags表格

$ php artisan make:model Taggable –m => 建立taggable中介表

    public function up()

    {

        Schema::create('posts', function (Blueprint $table) {

            $table->id();

            $table->string('name');

            $table->timestamps();

        });

    }

設置posts的欄位

    public function up()

    {

        Schema::create('videos', function (Blueprint $table) {

            $table->id();

            $table->string('name');

            $table->timestamps();

        });

    }

設置videos的欄位

    public function up()

    {

        Schema::create('tags', function (Blueprint $table) {

            $table->id();

            $table->string('name');

            $table->timestamps();

        });

    }

設置tags的欄位

    public function up()

    {

        Schema::create('taggables', function (Blueprint $table) {

            $table->id();

            $table->integer('tag\_id');

            $table->integer('taggable\_id');

            $table->string('taggable\_type');

            $table->timestamps();

        });

    }

設置taggable中介表的欄位

**Set up polymorphic many to many relationship**

class Post extends Model

{

    use HasFactory;

    public function tags() {

        $this->morphToMany('App\Models\Tag', 'taggable');

    }

}

在Post模型中建立與Tag模型的關係

class Video extends Model

{

    use HasFactory;

    public function tags() {

        $this->morphToMany('App\Models\Tag', 'taggable');

    }

}

在Video模型中建立與Tag模型的關係

**Create Polymorphic Many to Many Data**

Route::get('/insert', function() {

    $post = Post::create(['name'=>'First post']);

    $tag1 = Tag::findOrFail(1);

    $post->tags()->save($tag1);

    $video = Video::create(['name'=>'Video.mp4']);

    $tag2 = Tag::findOrFail(2);

    $video->tags()->save($tag2);

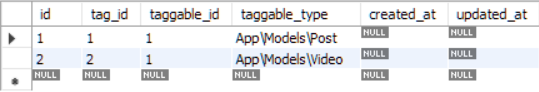
});

先建立一筆新的post資料，並且直接儲存進posts表格中 => 使用new Post的話要先save()才能做後續動作

$tag1儲存tags表格中id=1的資料

$post關聯tags，並且將這兩筆資料存進taggable的中介表中

video做一樣的動作



成功後taggables會有兩筆多型關聯的資料

**Reading Polymorphic Many to Many Data**

Route::get('/read', function() {

    $post = Post::findOrFail(1);

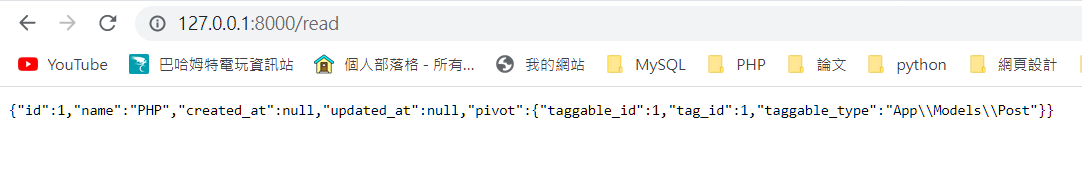
    foreach($post->tags as $tag) {

        return $tag;

    }

});

透過post與tag的關聯，找出post相關的tag有哪些



Route::get('/read', function() {

    $video = Video::findOrFail(1);

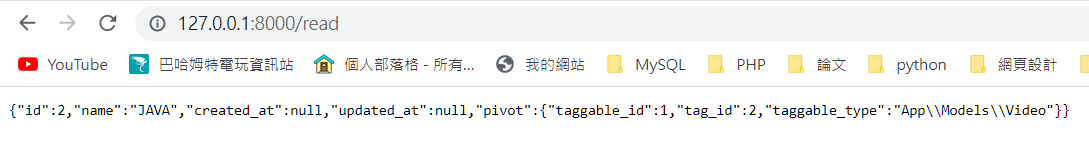
    foreach($video->tags as $tag) {

        return $tag;

    }

});

與post做的動作一樣



Route::get('/read', function() {

    $tag = Tag::findOrFail(1);

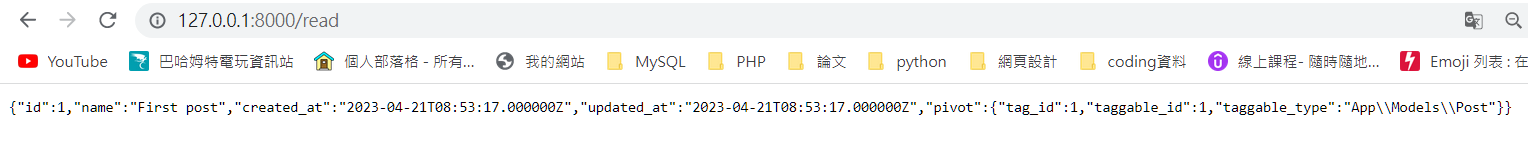
    foreach($tag->posts as $post) {

        return $post;

    }

});

有在Tag的模型中設置反向搜尋的功能，因此可以透過tags的資料去找出有哪些post或video



**Update Polymorphic Many to Many Data**

Route::get('/update', function() {

    $post = Post::findOrFail(1);

    foreach($post->tags as $tag) {

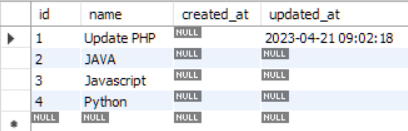
        $tag->name = 'Update PHP';

        $tag->save();

    }

});

與read資料類似，先找出post，透過post與tag之間的關聯，去修改tag的內容，再同步到資料庫中



Route::get('/update', function() {

    $post = Post::findOrFail(1);

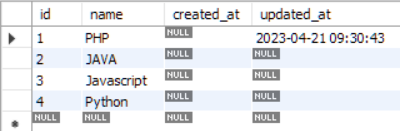
    foreach($post->tags as $tag) {

        $tag->whereName('Update PHP')->update(['name'=>'PHP']);

    }

});

另一種方式，使用條件式搜尋tag資料，再用update()將資料進行更新



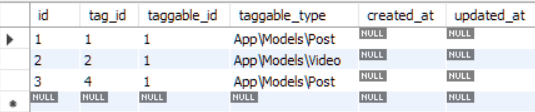
Route::get('/update', function() {

    $post = Post::findOrFail(1);

    $tag = Tag::findOrFail(4);

    $post->tags()->save($tag);

});



Route::get('/update', function() {

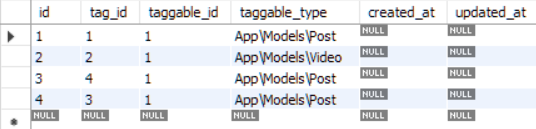
    $post = Post::findOrFail(1);

    $tag = Tag::findOrFail(3);

    $post->tags()->attach($tag);

});

另一種關聯的方式 attach()



Route::get('/update', function() {

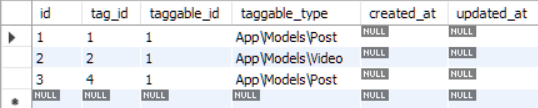
    $post = Post::findOrFail(1);

    $tag = Tag::findOrFail(3);

    $post->tags()->detach($tag);

});

detach()將先前建立的關聯給刪除



Route::get('/update', function() {

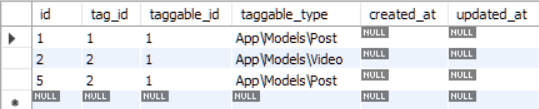
    $post = Post::findOrFail(1);

    $tag = Tag::findOrFail(3);

    $post->tags()->sync([1,2]);

});

sync()，post\_id=1的資料會連結tag\_id=1,2的資料，其他post->tag的資料都會被刪除



**Delete Polymorphic Many to Many Data**

Route::get('/delete', function() {

    $post = Post::findOrFail(1);

    foreach($post->tags as $tag) {

        $tag->whereId(4)->delete();

    }

});

透過post去搜尋tag，並且刪除tags中id=4的資料

