

TP N°9

Eloquent:Relationships

ISTA TINGHIR

Module: M205

Réalisée par: Groupe 9

le: 24 mai 2023

Développer en Back-End

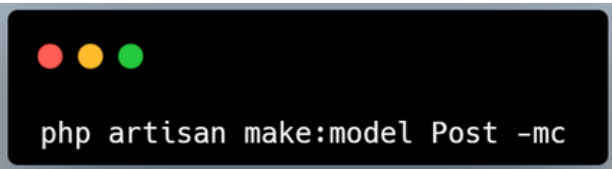
Introduction

Eloquent Relationships est une fonctionnalité de Laravel qui permet de définir et de gérer les relations entre les modèles dans une base de données. Eloquent est le système ORM (Object-Relational Mapping) de Laravel, qui facilite l'interaction avec la base de données en utilisant une syntaxe orientée objet.

Commençons l'exercice, êtes-vous prêts ?

Rappel !!

Pour créer un modèle, migration et aussi controller nous utilisons la commande :



```
php artisan make:model Post -mc
```

Création d'une migration avec clé étrangère (foreign key) :



```
Schema::create('posts', function (Blueprint $table) {  
    $table->id();  
    $table->string('title');  
    $table->text('content');  
    $table->unsignedBigInteger('user_id');  
    $table->timestamps();  
    $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');  
    // $table->foreignId('user_id')->constrained('users')->onUpdate('cascade')->onDelete('cascade');  
});
```

N'oubliez pas d'exécuter la commande :

```
`php artisan migrate`
```

1 . Les types de relations Eloquent

One-to-One Relationship :

Un enregistrement d'une table est lié à un seul enregistrement d'une autre table. Par exemple, un utilisateur peut avoir un seul profil.

Pour créer une relation "one-to-one" entre deux tables dans Laravel, suivez ces étapes :

1. Utilisez **php artisan make:migration** pour créer une migration pour chaque table.
2. Dans la première migration, ajoutez les colonnes nécessaires pour la première table à l'aide de la méthode **Schema::create**.
3. Dans la deuxième migration, ajoutez les colonnes nécessaires pour la deuxième table à l'aide de la méthode **Schema::create**, et ajoutez la clé étrangère faisant référence à la première table à l'aide de la méthode **foreign**.
4. Définir les relations dans les modèles :

```
class User extends Model
{
    public function phone()
    {
        return $this->hasOne(Phone::class);
    }
}
```

```
class Phone extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

5. Exécutez la commande **'php artisan migrate'**

Utilisation de la relation :

```
Route::get('/', function () {
    $profile = User::find(1)->profile;
    $user = Profile::find(1)->user;

    echo $user; // find user from Profile Model
    echo $profile; // find Profile from User Model
});
```

One To Many Relationship :

Un enregistrement d'une table peut être lié à plusieurs enregistrements d'une autre table. Par exemple, un utilisateur peut avoir plusieurs Posts.

Pour créer une relation "one-to-many" entre deux tables dans Laravel, Suivez les mêmes trois étapes que pour la relation "one-to-one" et également :

Définir les relations dans les modèles :

User Model :

```
class User extends Model
{
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

Post Model :

```
class Post extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Utilisation de la relation :

```
Route::get('/', function () {  
  
    $user = User::find(1); // Assuming user with ID 1  
    $posts = $user->posts;  
  
    foreach ($posts as $post) {  
        echo $post->title;  
        echo $post->content;  
        // Perform other operations with each post  
    }  
});
```

Has One OF Many Relationship :

Elle représente généralement une situation où un enregistrement dans une table peut être associé à un seul enregistrement parmi plusieurs options dans une autre table.

Voici les étapes pour créer une relation "Has One of Many" :

Étape 1 : Déclaration de la méthode dans le modèle parent


```
class Product extends Model  
{  
  
    public function latestOrder()  
    {  
  
        return $this->hasOne(Order::class)->latestOfMany();  
    }  
}
```

Étape 2 : Déclaration de la méthode inverse dans le modèle enfant (facultatif)

```
class Order extends Model  
{  
  
    public function parent()  
    {  
  
        return $this->belongsTo(Product::class);  
    }  
}
```

Étape 3 : Utilisation de la relation

```
Route::get('/', function () {  
    $product = Product::find(1);  
    $latestOrder = $product->latestOrder;  
    if ($latestOrder) {  
        echo "Latest Order Number: " . $latestOrder->order_number;  
    } else {  
        echo "No orders found.";  
    }  
});
```

[Read More!!](#) 

has-one-through Relationship :

has-many-through Relationship :

Many-to-Many Relationship :

Dans une relation **Many-To-Many**, un enregistrement dans une table peut être associé à plusieurs enregistrements dans une autre table, et vice versa. Pour représenter cette relation, vous aurez généralement besoin de trois tables de base de données : les deux tables liées et une table **pivot** qui les connecte.

Prenons un exemple de cette relation entre les tables **users** et **roles**. Chaque utilisateur peut avoir plusieurs rôles et chaque rôle peut être assigné à plusieurs utilisateurs.

1. Les tables nécessaires dans votre base de données :

```
users  
id - integer  
name - string
```

```
roles  
id - integer  
name - string
```

```
role_user(table pivot)  
user_id - integer  
role_id - integer
```

Après la création de ces trois tables, n'oubliez pas d'exécuter la commande :

'php artisan migrate'

2. Définition des modèles :

```
class User extends Model
{
    public function roles(): BelongsToMany
    {
        return $this->belongsToMany(Role::class, 'role_user');
    }
}
```

```
class Role extends Model
{
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class, 'role_user');
    }
}
```

3. Utiliser la relation :

```
Route::get('/', function () {
    $user = User::find(1);
    $rolesIds = [1,2,3];
    $user->roles()->attach($rolesIds);

    // replace attach() method with any one of this methods as u need :
    // sync() method remove prev data and add new giving data
    // syncWithoutDetach() keep prev data if exist and add new giving data
    // detach() remove giving data if exist in database

    return "attached";
});
```

4. Récupérer les données d'utilisateur avec son rôle :

```
Route::get('/', function () {
    return $user->load("roles"); // fetch user with it's role
});
```

[Read More!!](#) 

2. Polymorphic Relationships

Définition :

une relation polymorphique est une méthode de modélisation de données qui permet à une entité d'être associée à plusieurs autres entités de types différents. Cela offre une flexibilité dans la gestion des associations, car une seule colonne de clé étrangère est utilisée pour établir les liens avec différentes entités. Les relations polymorphiques simplifient la structure des données et rendent les associations plus flexibles, ce qui facilite la manipulation et la récupération des données dans le cadre du développement d'applications. Elles favorisent également la réutilisation du code et la facilité de maintenance.

Les types de relations polymorphique :

one-to-one polymorphic relation

Une relation polymorphique **One-To-One** est un type de relation dans une base de données où un enregistrement dans une table peut être associé à un seul enregistrement dans une autre table, et cette association est déterminée par une relation polymorphique.

Supposons que nous ayons un modèle "Media" qui représente une image, et nous voulons associer des images à différents types d'entités tels que les utilisateurs, les produits ou les articles. Voici comment vous pouvez l'implémenter :

1. Créez une migration pour la table "media" en utilisant la commande artisan :

```
php artisan make:migration create_media_table
```

2. Ouvrez la migration créée et ajoutez les colonnes nécessaires.

```
Schema::create('media', function (Blueprint $table) {  
    $table->id();  
    $table->string('url');  
    $table->unsignedBigInteger('mediable_id');  
    $table->string('mediable_type');  
    $table->timestamps();  
});
```

3. Exécutez la commande :

```
php artisan migrate
```

4. Créez un modèle "Media" :

```
class Media extends Model
{
    public function mediable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

5. Maintenant, pour chaque modèle qui peut avoir une image associée, vous devez déclarer la relation polymorphe inverse.

```
class User extends Model
{
    public function media(): MorphOne
    {
        return $this->morphOne(Media::class, 'mediable');
    }
}

class Product extends Model
{
    public function media(): MorphOne
    {
        return $this->morphOne(Media::class, 'mediable');
    }
}
```

6. Maintenant, vous pouvez utiliser la relation pour associer une image à une entité

```
Route::get("/morph", function () {
    $user = User::find(1);
    $user->media()->create([
        'filename' => 'image.jpg',
        'path' => '/path/to/image.jpg',
    ]);
});
```

7. Pour la récupération :

```
Route::get("/", function () {
    $user = User::find(1);
    $userImage = $user->media;
    return $userImage;
});
```


one-to-many polymorphic relation

La relation polymorphique de type **one-to-many** permet à un modèle d'être associé à plusieurs instances d'autres modèles, tout en permettant à ces autres modèles d'être associés à d'autres modèles également. Cela permet une relation flexible entre différents types d'entités.

supposons que les utilisateurs de votre application peuvent laisser des commentaires à la fois sur des publications (posts) et des vidéos. En utilisant des relations polymorphiques, vous pouvez utiliser une seule table "comments" pour stocker les commentaires pour les publications et les vidéos. Voici comment vous pouvez l'implémenter :

1. La structure de table `comments`

```
Schema::create('comments', function (Blueprint $table) {
    $table->id();
    $table->text('body');
    $table->unsignedBigInteger('commentable_id');
    $table->string('commentable_type');
    $table->timestamps();
});
```

2. Définissez les modèles et les relations :

```
class Comment extends Model
{
    public function commentable(): MorphTo
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

class Video extends Model
{
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

3. Création d'un commentaire pour une publication (post) 4. Récupération (all comments of post)

```
Route::get("/morphOTM", function () {
    $post = Post::find(1);
    $comment = new Comment([
        'body' => 'Un commentaire sur la publication'
    ]);
    $post->comments()->save($comment);
});
```

```
Route::get("/recuperation", function () {
    $post = Post::find(1);
    $comments = $post->comments();
});
```

one-of-many polymorphic relation

Parfois, un modèle peut avoir de nombreux modèles associés, mais vous souhaitez récupérer facilement le modèle associé le plus récent ou le plus ancien de la relation.

Par exemple, un modèle User peut être associé à plusieurs modèles Image, mais vous souhaitez définir un moyen pratique d'interagir avec l'image la plus récente téléchargée par l'utilisateur. Vous pouvez y parvenir en utilisant le type de relation morphOne combiné avec les méthodes ofMany.

1. Dans votre modèle User, vous pouvez définir la relation morphOne de la manière suivante :

```
class User extends Model
{
    public function latestImage(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable')->latest();
    }
}
```

2. Maintenant, vous pouvez facilement récupérer l'image la plus récente téléchargée par un utilisateur :

```
$user = User::find(1); // Récupère l'utilisateur avec l'ID 1
$latestImage = $user->latestImage;
```

Many-To-Many polymorphic relation

Les relations polymorphiques de type plusieurs-à-plusieurs (many-to-many polymorphic relations) sont légèrement plus complexes que les relations "morph one" et "morph many". Par exemple, un modèle "Post" et un modèle "Video" pourraient partager une relation polymorphique avec un modèle "Tag". L'utilisation d'une relation polymorphique de type plusieurs-à-plusieurs dans cette situation permettrait à votre application d'avoir une seule table de tags uniques qui peuvent être associés à des publications (posts) ou des vidéos.

Tout d'abord, examinons la structure de table requise pour établir cette relation :

```
posts
id - integer
name - string
```

```
tags
id - integer
name - string
```

```
videos
id - integer
name - string
```

```
taggables
tag_id - integer
taggable_id - integer
taggable_type - string
```

Après la création de ces trois tables, n'oubliez pas d'exécuter la commande :

'php artisan migrate'

2.Définition des modèles :

```
class Post extends Model
{
    //add fillable
    public function tags(): MorphToMany
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

```
class Video extends Model
{
    //add fillable

    public function tags(): MorphToMany
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

```
class Tag extends Model
{
    //add fillable

    public function posts(): MorphToMany
    {
        return $this->morphedByMany(Post::class, 'taggable');
    }

    public function videos(): MorphToMany
    {
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

Utilisation de la relation dans notre code :

```
Route::get("/", function () {

    $post = Post::find(1);
    $tag = Tag::create(['name' => 'Laravel']);
    $post->tags()->attach($tag);
    return "attached";
});
```