Cours Laravel 6 – les données – la relation n:n

laravel.sillo.org/cours-laravel-6-les-donnees-la-relation-nn/

bestmomo

September 3, 2019

Dans le précédent chapitre nous avons vu la relation de type **1:n**, la plus simple et la plus répandue. Nous allons maintenant étudier la relation de type **n:n**, plus délicate à comprendre et à mettre en œuvre. Nous allons voir qu'Eloquent permet de simplifier la gestion de ce type de relation.

On va continuer l'application de gestion de films, toujours avec des catégories, mais maintenant on va considérer qu'un film peut appartenir à plusieurs catégories, ce qui change pas mal de choses...

Les données

La relation n:n

Imaginez une relation entre deux tables A et B qui permet de dire :

- je peux avoir une ligne de la table A en relation avec plusieurs lignes de la table B,
- je peux avoir une ligne de la table B en relation avec plusieurs lignes de la table A.

Cette relation ne se résout pas comme nous l'avons vu au chapitre précédent avec une simple clé étrangère dans une des tables. En effet il nous faudrait des clés dans les deux tables et plusieurs clés, ce qui n'est pas possible à réaliser.

La solution consiste à créer une table intermédiaire (nommée table **pivot**) qui sert à mémoriser les clés étrangères. On va donc toujours avoir nos tables **films** et **categories** mais en plus une table pivot entre les deux.

Les migrations

Au niveau des migration celle pour les catégories ne va pas changer. Pour celle des films on va retirer la clé étrangère, donc il ne va rester que ça :

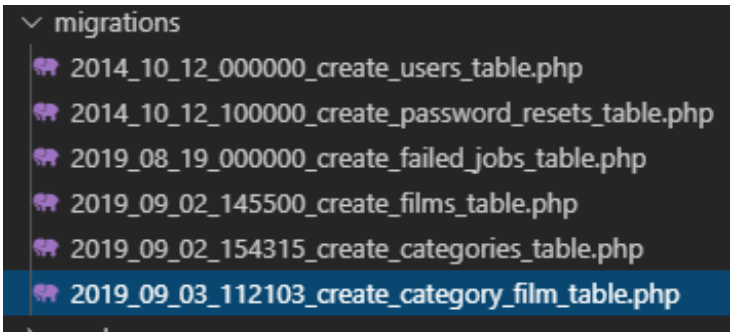
```
public function up()
{
   Schema::create('films', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->string('title');
      $table->year('year');
      $table->text('description');
      $table->timestamps();
      $table->softDeletes();
   });
}
```

Et il nous faut en plus la migration pour la table pivot qui elle aura 2 clés étrangère : une pour les catégories et une autre pour les films :

```
php artisan make:migration create_category_film_table
```

Par convention on met les deux noms au singulier et dans l'ordre alphabétique donc **category_film**.

Et on code la migration pour les deux clé étrangères :



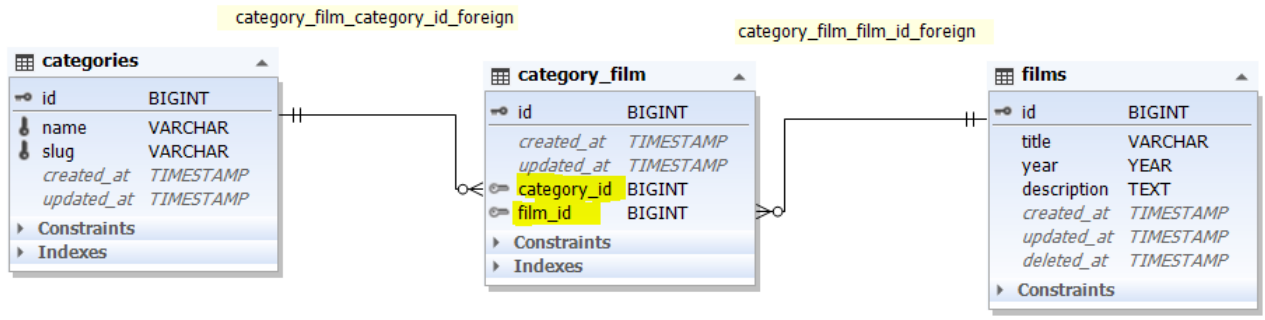

```
public function up()
{
   Schema::create('category_film',
function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->timestamps();

      $table-
>unsignedBigInteger('category_id');
      $table->foreign('category_id')
         ->references('id')
         ->on('categories')
         ->onDelete('cascade')
         ->onUpdate('cascade');

      $table->unsignedBigInteger('film_id');
      $table->foreign('film_id')
         ->references('id')
         ->on('films')
         ->onDelete('cascade')
         ->onUpdate('cascade');
   });
}
```

Cette fois j'ai choisi **cascade** pour les clés étrangères. Donc si on supprime une catégorie ou un film la table pivot sera automatiquement mise à jour.

Et on rafraichit la base :

```
php artisan migrate:fresh
```

Au niveau des tables on a ce schéma :



La relation entre les deux tables est assurée par la table pivot . Cette table pivot contient les clés des deux tables :

- **category_id** pour mémoriser la clé de la table **categories,**
- **film_id** pour mémoriser la clé de la table **films**.

De cette façon on peut avoir plusieurs enregistrements liés entre les deux tables, il suffit à chaque fois d'enregistrer les deux clés dans la table pivot. Évidemment au niveau du code ça demande un peu d'intendance parce qu'il y a une table supplémentaire à gérer.

Les modèles

Dans le modèle **Category** on change la relation :

```
public function films()
{
   return $this->belongsToMany(Film::class);
}
```

On déclare ici avec la méthode **films** (au pluriel) qu'une catégorie appartient à plusieurs (**belongsToMany**) films (Film). On aura ainsi une méthode pratique pour récupérer les films d'une catégorie.

C'est exactement pareil pour le modèle **Film** :

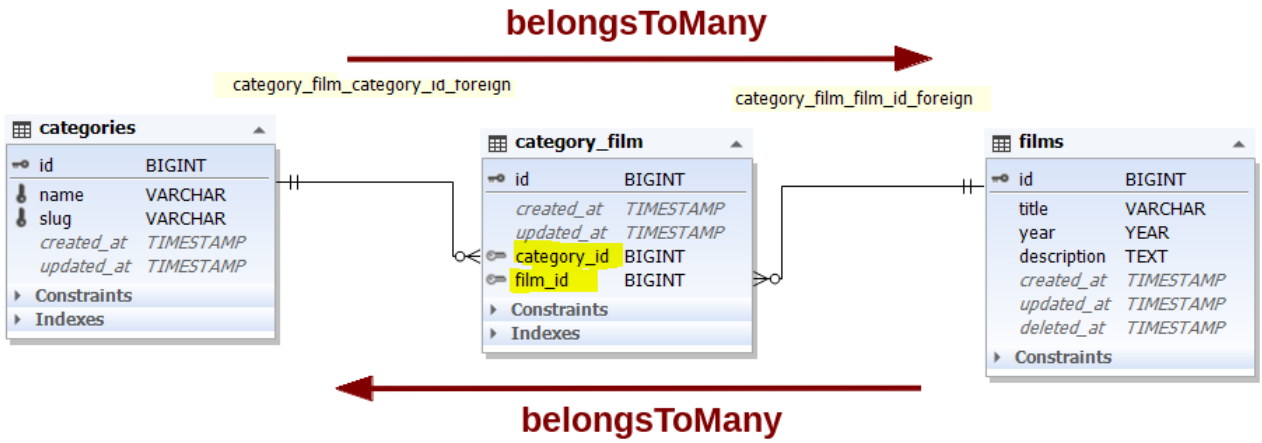
```
protected $fillable = ['title', 'year', 'description'];

public function categories()
{
   return $this->belongsToMany(Category::class);
}
```

Au passage j'ai supprimé la colonne **category_id** dans la propriété **$fillable**.

C'est le même principe que pour les catégories puisque la relation est symétrique. On déclare avec la méthode **categories** (au pluriel) qu'un film appartient à plusieurs (**belongsToMany**) catégories (**Category**).

La relation n:n

Voici une schématisation de cette relation avec les deux méthodes symétriques :



Toutes les méthodes qu'on a vues pour la relation **1:n** fonctionnent avec la relation **n:n**, ce qui est logique. Le fait qu'il y ait une table pivot ne change rien au fait que la relation, vue de l'une des deux tables, ressemble à s'y méprendre à une relation **1:n**. Si je choisis une catégorie par exemple je sais qu'elle peut avoir plusieurs films liés.

La population

Il nous faut encore créer des enregistrements pour nos essais. On ne peut pas se contenter de garder ce qu'on avait fait parce que maintenant ça serait bien qu'un film appartienne à plusieurs catégories. Alors voilà le nouveau code de **DatabaseSeeder** :

```
public function run()
{
   factory(App\Category::class, 10)->create();

   $ids = range(1, 10);

   factory(App\Film::class, 40)->create()->each(function ($film) use($ids) {
      shuffle($ids);
      $film->categories()->attach(array_slice($ids, 0, rand(1, 4)));
   });
}
```

On commence par créer 10 catégories. Ensuite on crée 40 films et pour chacun on attache entre 1 et 4 catégories. On passe à la méthode **attach** l'identifiant (on peut en mettre plusieurs dans un tableau comme je l'ai fait ici) de l'enregistrement en relation et Eloquent se charge de renseigner la table pivot. Il existe aussi la méthode **detach** qui fait exactement l'inverse.

film_id ▲ 1	category_id
1	9
1	4
2	10
2	9
2	3
2	5
3	8
4	6
5	7
5	1
5	9
6	9
6	1

Affichage d'un film et eager loading

Pour l'affichage d'un film on avait prévu de préciser la catégorie à laquelle appartenait ce film. Il est évident qu'il va falloir prendre en compte maintenant le fait qu'on peut avoir plusieurs catégories. On va mettre à jour la méthode show du contrôleur :

```
public function show(Film $film)
{
   $film->with('categories')->get();
   return view('show', compact('film'));
}
```

On sait qu'avec la liaison de la route on a déjà le modèle du film. On le complète en ajoutant (**with**) ses catégories. A la sortie on a une collection avec la relation :

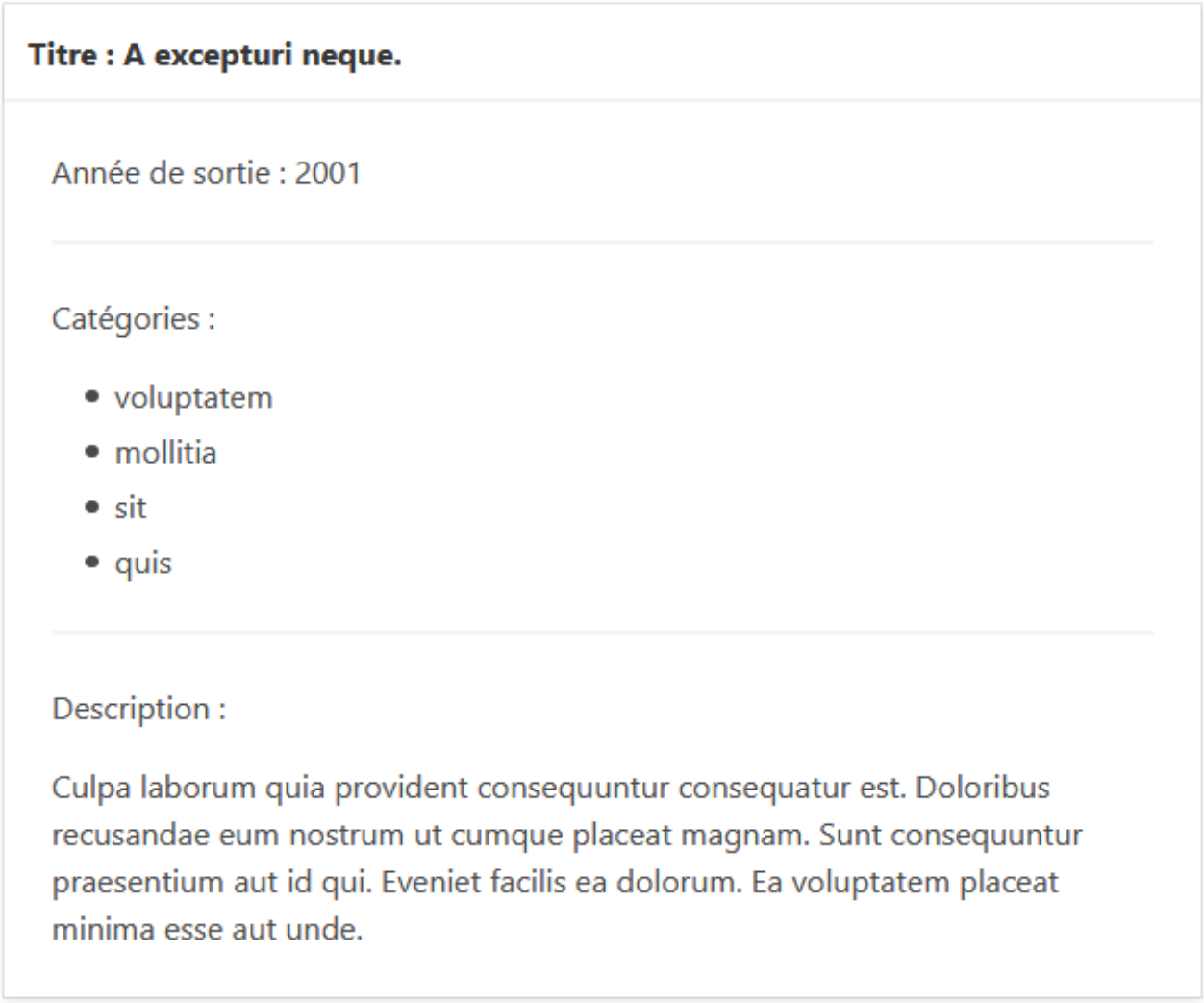
```
Collection {#265 ▼
  #items: array:40 [▼
    0 => Film {#276 ▼
      #fillable: array:3 [▼
        0 => "title"
        1 => "year"
        2 => "description"
      ]
      ...
      #attributes: array:7 [▼
        "id" => 1
        "title" => "In qui cumque."
        "year" => 2013
        "description" => "Consequatur cumque odit delectus velit et. Sit non qui harum vel quas autem
numquam. Repellat ea praesentium voluptas fugit hic. Voluptate ut tempore neque veni ▶"
        "created_at" => "2019-09-03 11:58:23"
        "updated_at" => "2019-09-03 11:58:23"
        "deleted_at" => null
      ]
      ...
      #relations: array:1 [▼
        "categories" => Collection {#317 ▼
          #items: array:2 [▼
            0 => Category {#452 ▶}
            1 => Category {#500 ▶}
          ]
        }
      ]
      ...
```

On appelle cette manière de faire l'eager loading (par opposition au lazy loading). Ça permet d'éviter de multiples accès à la base pour aller récupérer des valeurs.

Maintenant dans la vue show on n'a plus de problème pour afficher les catégories :

```
@extends('template')

@section('content')
   <div class="card">
      <header class="card-header">
         <p class="card-header-title">Titre : {{ $film->title }}</p>
      </header>
      <div class="card-content">
         <div class="content">
            <p>Année de sortie : {{ $film->year }}</p>
            <hr>
            <p>Catégories :</p>
            <ul>
               @foreach($film->categories as $category)
                  <li>{{ $category->name }}</li>
               @endforeach
            </ul>
            <hr>
            <p>Description :</p>
            <p>{{ $film->description }}</p>
         </div>
      </div>
   </div>
@endsection
```

On utilise la directive **@foreach** pour boucler sur les catégories :



Il y a quand même une chose qui me dérange. On a deux accès à la base :

- au niveau du traitement de la route avec la liaison implicite
- au niveau du contrôleur pour aller chercher les catégories

Ça serait quand même plus élégant de tout faire d'un coup ! On va changer la liaison implicite en liaison explicite. Dans le provider **RouteServiceProvider** qui, comme son nom l'indique, est consacré aux routes, on va ajouter ce code :

```
use App\Film;

...

public function boot()
{
    parent::boot();

    Route::bind('film', function ($value) {
        return Film::with('categories')->find($value) ?? abort(404);
    });
}
```

Maintenant on peu revenir à la version initiale du code dans le contrôleur :

```
public function show(Film $film)
{
   return view('show', compact('film'));
}
```

On a ainsi un seul accès à la base ! Évidemment on aura le chargement des catégories pour toutes les routes concernées, mais ce n'est pas gênant et pourra même s'avérer très utile ! Sauf pour le soft delete où ça va coincer. Alors pour ces deux routes on va changer le nom du paramètre :

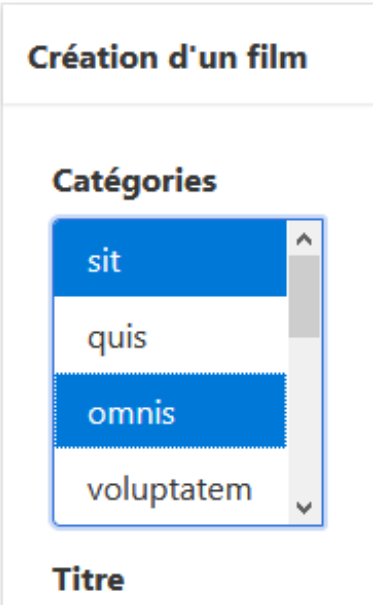
```
Route::delete('films/force/{id}', 'FilmController@forceDestroy')->name('films.force.destroy');
Route::put('films/restore/{id}', 'FilmController@restore')->name('films.restore');
```

Création d'un film

Pour la création d'un film on va aussi devoir modifier le code parce qu'on ne peut choisir qu'une catégorie dans la liste. On va transformer la liste dans la vue **create** pour un choix multiple :

```
<label class="label">Catégories</label>
<div class="select is-multiple">
   <select name="cats[]" multiple>
      @foreach($categories as $category)
         <option value="{{ $category->id }}" {{ in_array($category-
>id, old('cats') ?: []) ? 'selected' : '' }}>{{ $category->name }}</option>
      @endforeach
   </select>
</div>
```

Remarquez que le nom du **select** est accompagné de crochets (**cats[]**) pour signifier qu'on va envoyer un tableau de valeurs. Remarquez aussi la stratégie pour récupérer les catégories sélectionnées en cas de souci de validation.



On va avoir un peu plus de travail dans la méthode **store** du contrôleur.

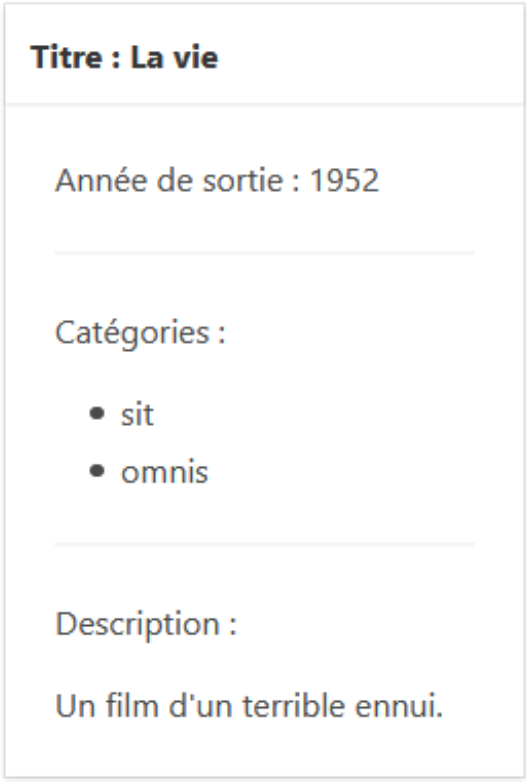
On a dans la requête ces éléments :

```
array:5 [▼
  "_token" => "qqARcLpGc6YDJ4jRpzazHeDbPlrxMSnSZYbtO9hJ"
  "cats" => array:2 [▼
    0 => "1"
    1 => "3"
  ]
  "title" => "La vie"
  "year" => "1952"
  "description" => "Un film d'un terrible ennui."
]
```

Voici la méthode **store** modifiée en conséquence :

```
public function store(FilmRequest $filmRequest)
{
   $film = Film::create($filmRequest->all());
   $film->categories()->attach($filmRequest->cats);
   return redirect()->route('films.index')->with('info', 'Le film a bien été créé');
}
```

On vérifie qu'on a les bonnes catégories dans la fiche du film :



Modification d'un film

Ce qu'on a fait pour la création va nous servir pour la modification. D'ailleurs précédemment on n'avait pas prévu la modification de la catégorie pour un film pour ne pas trop alourdir le code, mais là on va le faire.

Déjà dans le provider **AppServiceProvider** on va ajouter la vue **edit** dans le composeur de vue pour récupérer toutes les catégories :

```
public function boot()
{
   View::composer(['index', 'create', 'edit'], function ($view) {
      $view->with('categories', Category::all());
   });
}
```

Ensuite dans la vue **edit** on ajoute la liste des catégories comme on l'a fait pour la vue **create** :

```
<div class="field">
   <label class="label">Catégories</label>
   <div class="select is-multiple">
      <select name="cats[]" multiple>
         @foreach($categories as $category)
            <option value="{{ $category->id }}" {{ in_array($category->id, old('cats') ?: $film->categories->pluck('id')->toArray()) ? 'selected' : '' }}>{{ $category->name }}</option>
         @endforeach
      </select>
   </div>
</div>
```

Le remplissage est un peu plus délicat parce qu'on a deux situations :

- quand on charge le formulaire au départ on doit sélectionner les catégories actuelles du film
- quand on a un retour de validation incorrecte on doit sélectionner les catégories précédemment sélectionnées

Laravel possède un système de collections très performant. Par exemple ici quand on écrit **$film->categories** on obtient la collection de toutes les catégories du film. La méthode **pluck** permet de ne garder que la clé qui nous intéresse, ici l'**id**. Enfin la méthode **toArray** transforme la collection en tableau parce que c'est ce que nous avons besoin ici.

Il ne nous reste plus qu'à modifier la méthode **update** du contrôleur :

```
public function update(FilmRequest $filmRequest, Film $film)
{
   $film->update($filmRequest->all());
   $film->categories()->sync($filmRequest->cats);
   return redirect()->route('films.index')->with('info', 'Le film a bien été modifié');
}
```

Pour modifier la table pivot on utilise cette fois la puissante méthode **sync**. On lui donne un tableau en paramètre, si un enregistrement se trouve dans la table mais pas dans le tableau alors il est supprimé et si une valeur se trouve dans la tableau mais pas dans la table alors on crée l'enregistrement dans la table. C'est bien une synchronisation !

On a maintenant une application qui commence à ressembler à quelque chose !

J'ai prévu un ZIP récupérable ici qui contient le code de cet article.

En résumé

- Une relation de type **n:n** nécessite la création d'une table pivot.
- L'eager loading permet de limiter le nombre d'accès à la base de données.
- Si la liaison implicite ne suffit pas on peut faire de la liaison explicite avec un traitement personnalisé.
- Eloquent gère élégamment les tables pivots avec des méthodes adaptées (attach,

detach, sync...).