Cours Laravel 6 – les données – le polymorphisme

laravel.sillo.org/cours-laravel-6-les-donnees-le-polymorphisme/

bestmomo

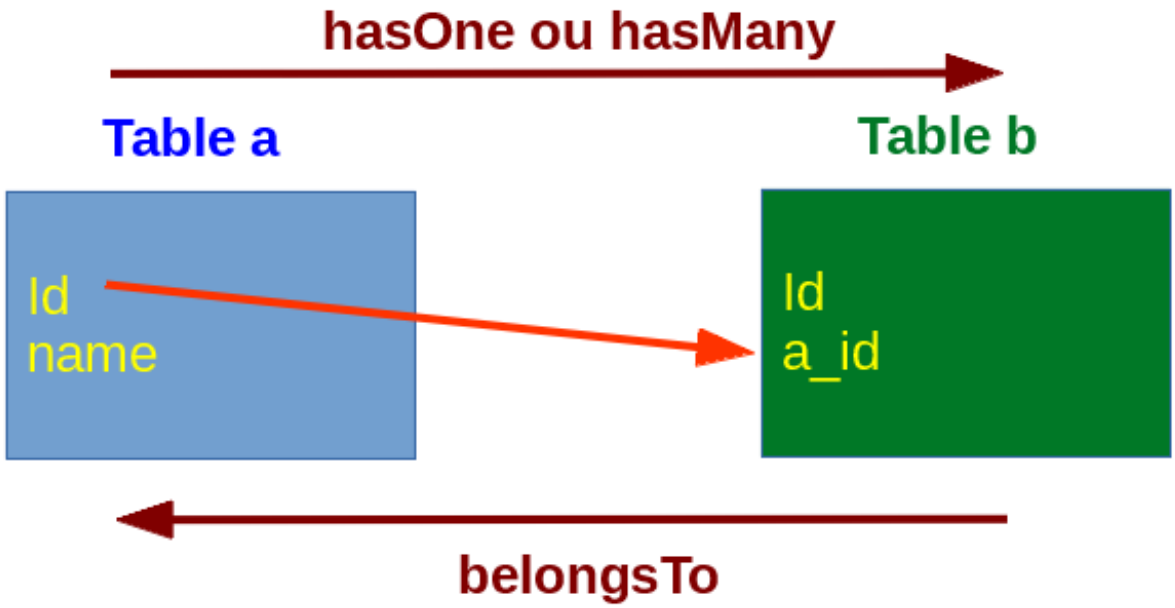
September 4, 2019

Lors des deux précédents chapitres on a vu les principales relations que nous offre Eloquent : **hasMany** et **belongsToMany**. Je ne vous ai pas parlé de la relation **hasOne** parce que c'est juste du **hasMany** limité à un seul enregistrement et est peu utilisé. Dans tous les cas qu'on a vus on considère 2 tables en relation. Dans le présent chapitre on va envisager le cas où une table peut être en relation avec plusieurs autres tables, ce qui se nomme du polymorphisme.

Un peu de théorie

La relation 1:1 ou 1:n

On a vu cette relation, en voici une schématisation pour fixer les esprits :

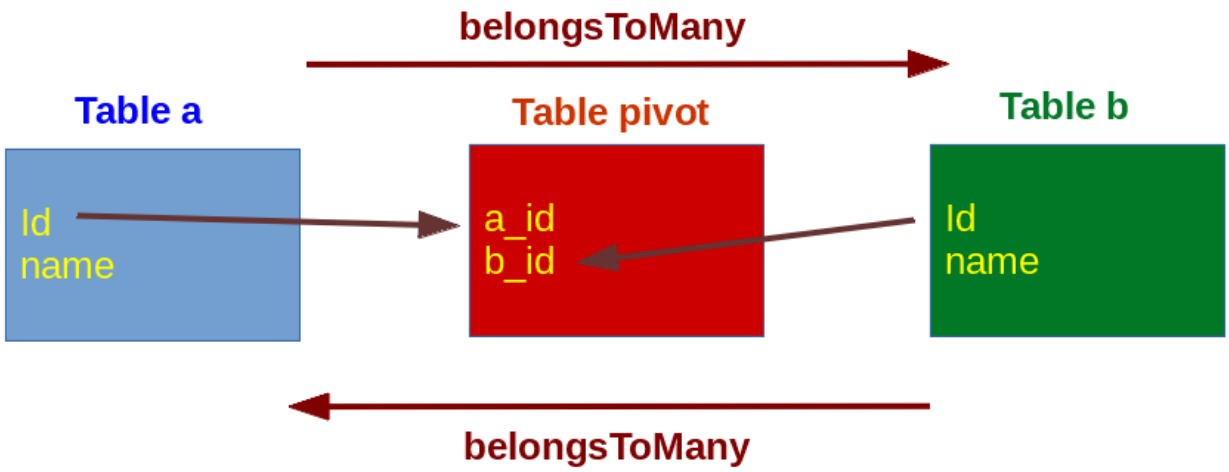


On a **a possède un b** (**hasOne**) ou **a possède plusieurs b** (**hasMany**).

La réciproque : **b est possédé par un a** (**belongsTo**).

La relation n:n

On a vu aussi cette relation, en voici une schématisation pour fixer les esprits :



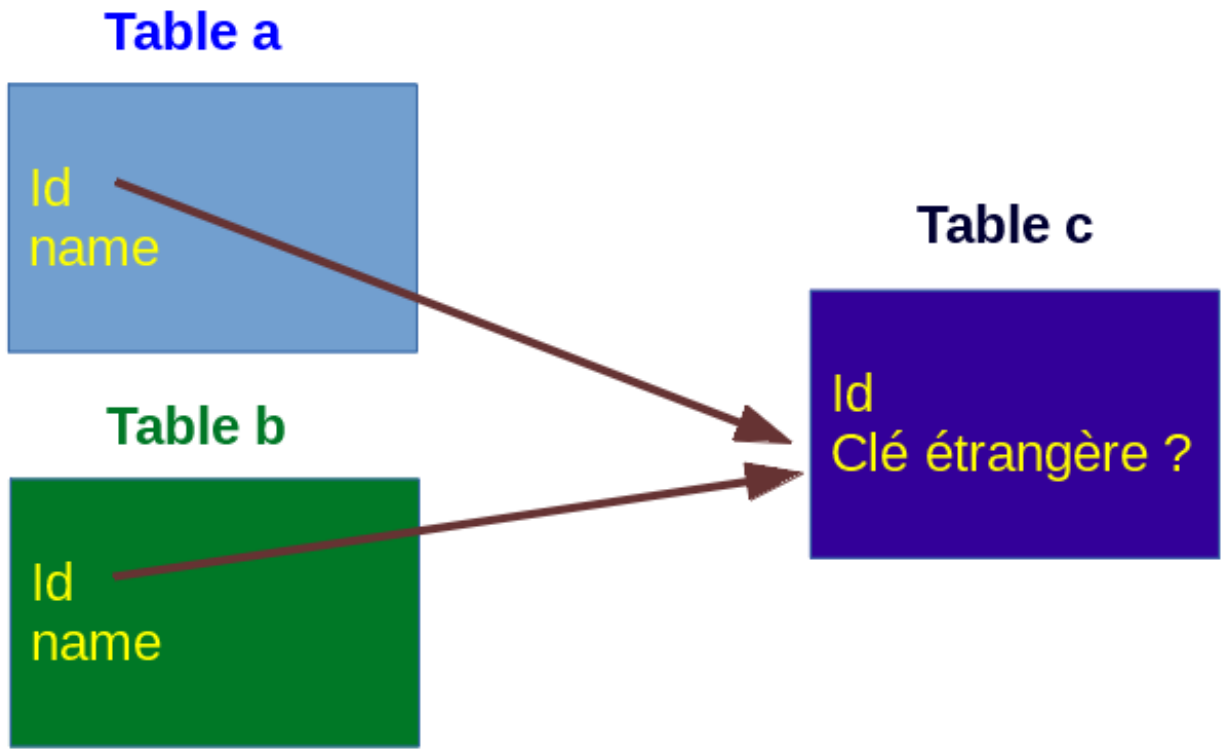
On a **a appartient à un ou plusieurs b** (**belongsToMany**).

Et on a **b appartient à un ou plusieurs a** (**belongsToMany**).

La relation une table vers plusieurs tables

Type de relation 1:n

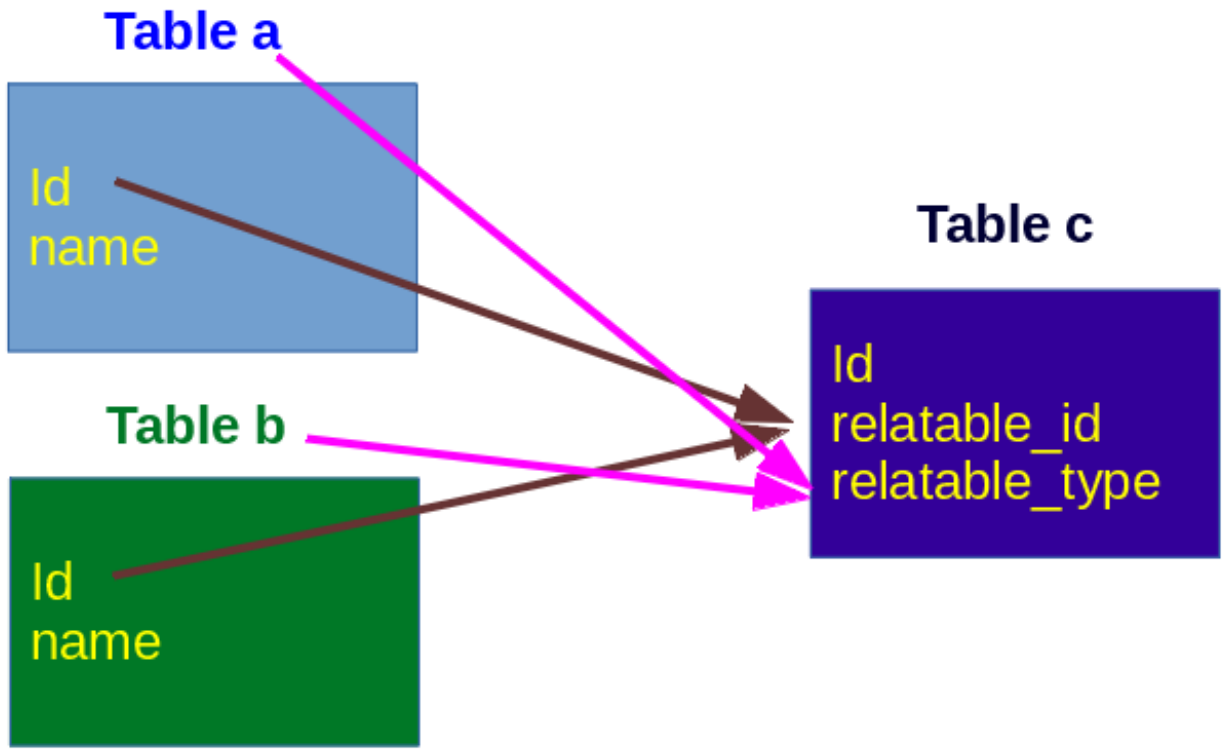
Maintenant imaginons cette situation :



La table **c** peut être en relation soit avec la table **a**, soit avec la table **b**. Dans cette situation comment gérer une clé étrangère dans la table c ? Comment l'appeler et comment savoir avec quelle table elle est en relation ?

On voit bien qu'il va falloir une autre information : **connaître sûrement la table en relation**.

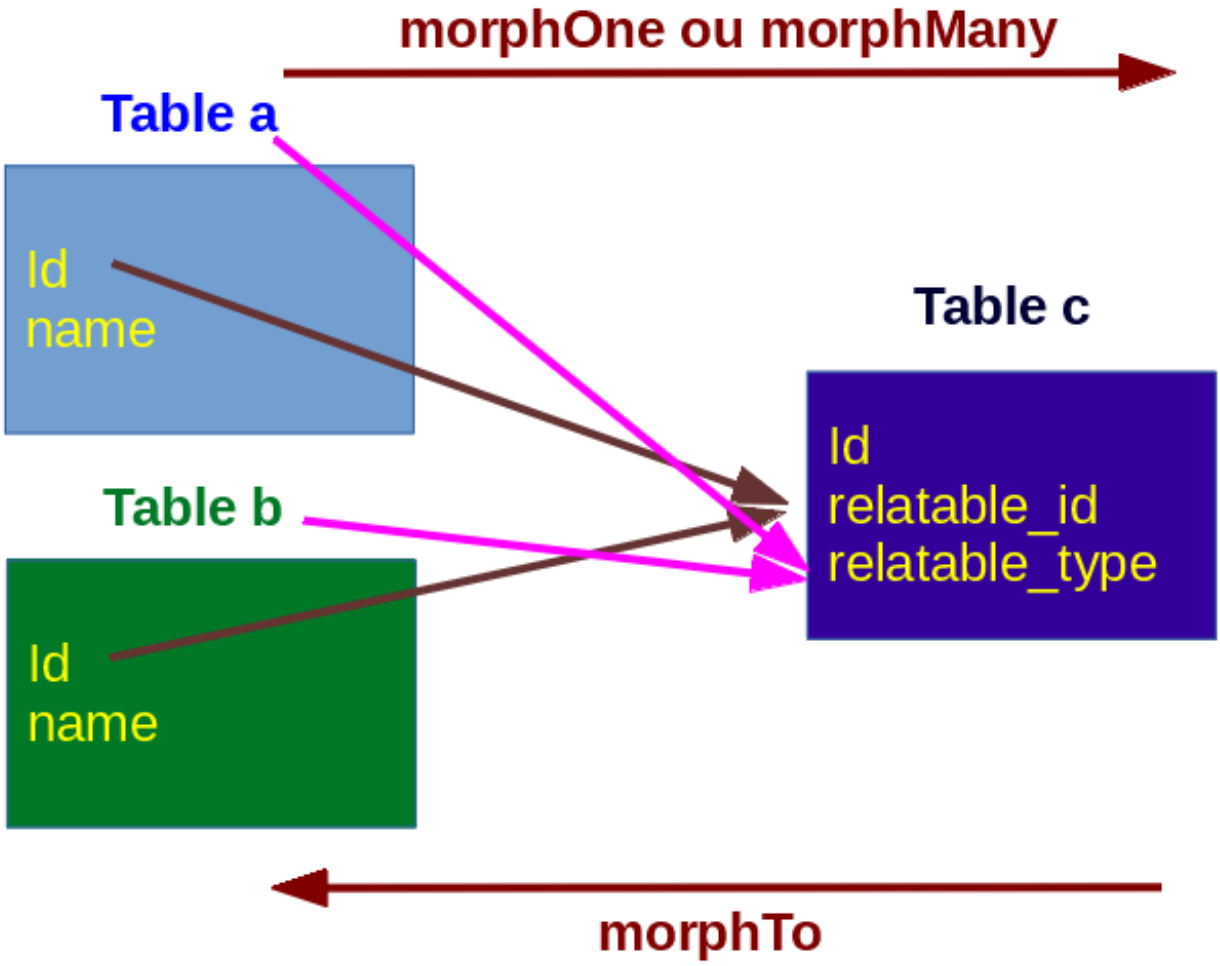
Puisqu'on a besoin de deux informations il nous faut deux colonnes :



On a donc deux colonnes :

- **relatable_id** : la clé étrangère qui mémorise l'identifiant de l'enregistrement en relation
- **relatable_type** : la classe du modèle en relation.

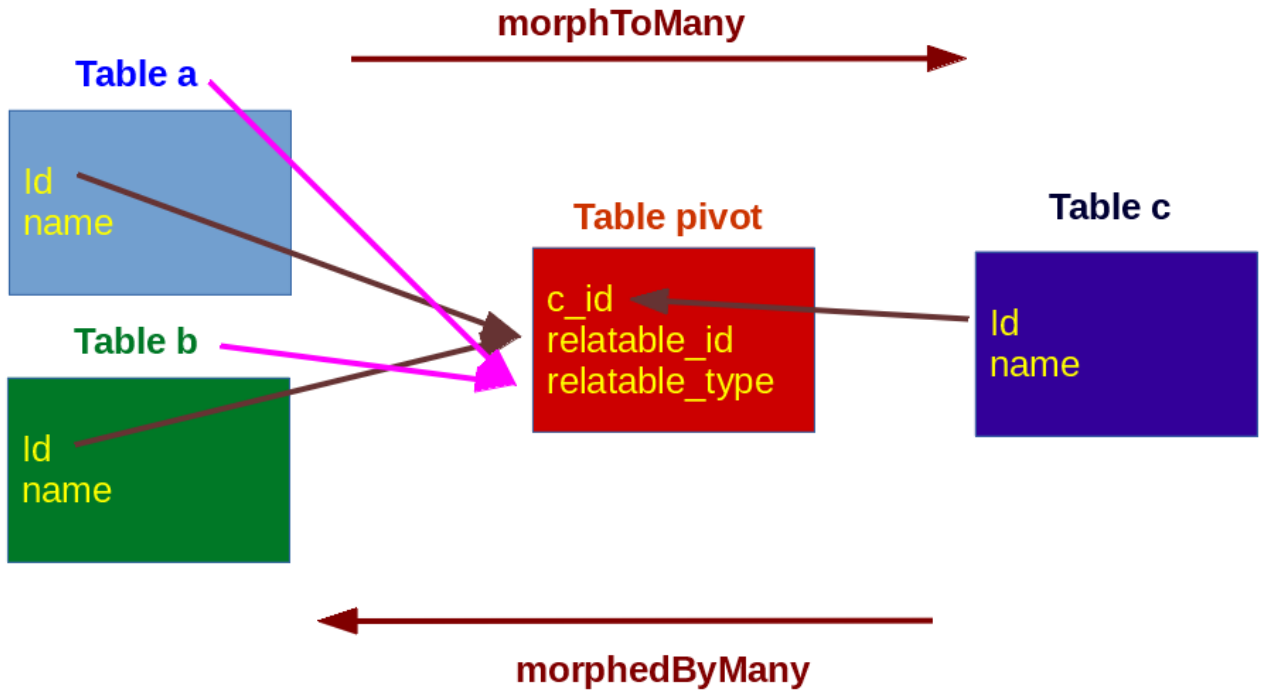
Voici la figure complétée avec les noms de ces relations :



- **morphOne** : c'est le **hasOne** mais issu de plusieurs tables.
- **morphMany** : c'est le **hasMany** mais issu de plusieurs tables.
- **morphTo** : c'est le **belongsTo** mais à destination de plusieurs tables.

Type de relation n:n

On peut avoir le même raisonnement pour une relation de type **n:n** avec plusieurs tables d'un côté de la relation :



- **morphToMany** : c'est le **belongsToMany** mais issu de plusieurs tables.
- **morphedByMany** : c'est le **belongsToMany** mais en direction de plusieurs tables.

Les données

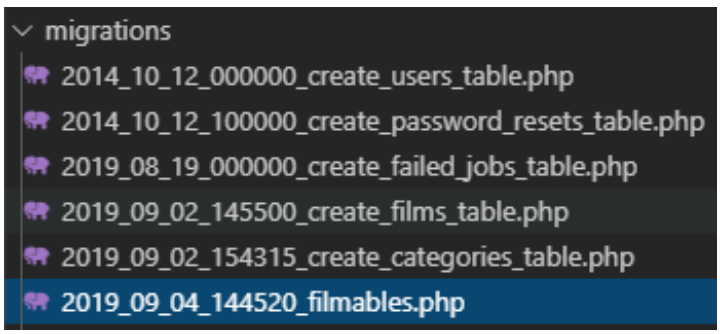
On va poursuivre l'exemple de la gestion des films en introduisant du polymorphisme. Jusque là on avait des catégories et des films, on va ajouter des acteurs, c'est quand même assez pertinent pour des films ! Un acteur peut jouer dans plusieurs films et un film a plusieurs acteurs, on a donc à nouveau une relation de type **n:n**. On pourrait évidement utiliser une seconde table pivot mais ça sera plus élégant avec une seule et du polymorphisme.

Les migrations

On ne change rien aux migrations des catégories et des films. On supprime la table pivot qu'on avait créée. Et on crée une migration pour notre nouvelle table pivot :

```
php artisan make:migration filmables
```

Et on va prévoie ce code :



```php
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class Filmables extends Migration
{
    public function up()
    {
        Schema::create('filmables', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
            $table->unsignedBigInteger('film_id');
            $table->foreign('film_id')
                ->references('id')
                ->on('films')
                ->onDelete('cascade')
                ->onUpdate('cascade');
            $table->morphs('filmable');
        };
    }

    public function down()
    {
        Schema::dropIfExists('filmable');
    }
}
```

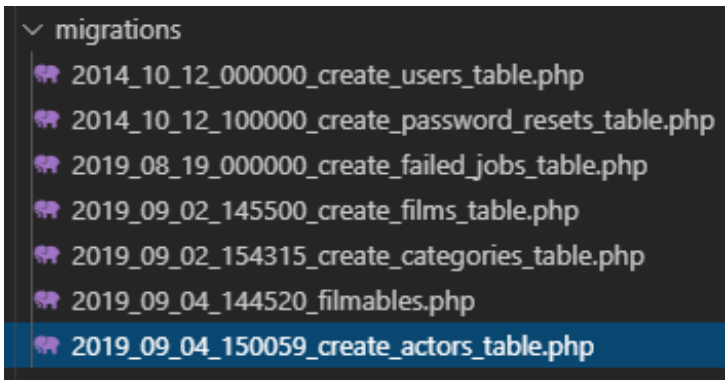
La méthode **morphs** est bien pratique parce qu'elle crée automatiquement les deux colonnes pour la relation polymorphique.

On a aussi besoin d'une migration et d'un modèle pour les acteurs :

```
php artisan make:model Actor --migration
```

Avec ce code :

```php
Schema::create('actors', function
(Blueprint $table) {
    $table->bigIncrements('id');
    $table->string('name')->unique();
    $table->string('slug')->unique();
    $table->timestamps();
});
```



On prévoit juste un nom et un slug, comme pour les catégories.

On régénère les tables :

```
php artisan migrate:fresh
```

Si tout se passe bien on doit avoir les 4 tables qui nous intéressent :

Les relations

Category

Pour le modèle **Category** on prévoit cette relation (qui remplace la précédente) :

```
protected $fillable = ['name', 'slug'];

public function films()
{
    return $this->morphToMany(Film::class, 'filmable');
}
```

Actor

Ca va être pareil pour le modèle **Actor** :

```
public function films()
{
    return $this->morphToMany(Film::class, 'filmable');
}
```

Film

Pour le modèle **Film** on prévoit ces deux relations :

```
public function categories()
{
   return $this->morphedByMany(Category::class, 'filmable');
}

public function actors()
{
   return $this->morphedByMany(Actor::class, 'filmable');
}
```

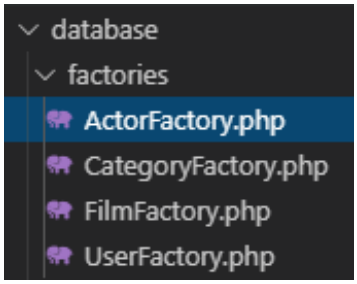
La population

On va remplir les tables pour nos essais.

On va avoir besoin d'un factory pour les acteurs :

```
php artisan make:factory ActorFactory
```

Avec ce code :




```
<?php

use App\Actor;
use Faker\Generator as Faker;
use Illuminate\Support\Str;

$factory->define(Actor::class, function (Faker $faker) {
   $name = $faker->name();
   return [
      'name' => $name,
      'slug' => Str::slug($name),
   ];
});
```

Il ne reste plus qu'à modifier le code de **DatabaseSeeder** :

```php
<?php

use Illuminate\Database\Seeder;
use Illuminate\Support\Str;
use App\Category;

class DatabaseSeeder extends Seeder
{
    public function run()
    {
        factory(App\Actor::class, 10)->create();

        $categories = [
            'Comédie',
            'Drame',
            'Action',
            'Fantastique',
            'Horreur',
            'Animation',
            'Espionnage',
            'Guerre',
            'Policier',
            'Pornographique',
        ];

        foreach($categories as $category) {
            Category::create(['name' => $category, 'slug' => Str::slug($category)]);
        }

        $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)));
            shuffle($ids);
            $film->actors()->attach(array_slice($ids, 0, rand(1, 4)));
        });
    }
}
```

Cette fois j’ai prévu des noms de catégories réalistes et non plus aléatoires :

J'ai aussi prévu 10 acteurs avec des noms aléatoires :

On a aussi 40 films. Et enfin dans la table pivot j'ai créé aussi de façon aléatoire des liaisons entre catégories et acteurs et films. En voici un aperçu :

On voit que pour chaque enregistrement on a le nom du modèle et l'id comme on l'avait prévu.

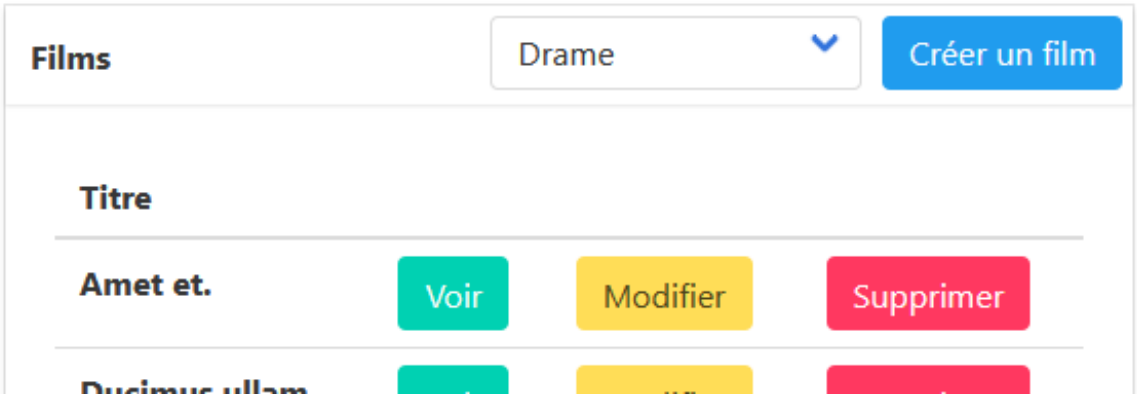
id	name	slug
1	Comédie	comedie
2	Drame	drame
3	Action	action
4	Fantastique	fantastique
5	Horreur	horreur
6	Animation	animation
7	Espionnage	espionnage
8	Guerre	guerre
9	Policier	policier
10	Pornographique	pornographique

id	name	slug
1	Chyna Bogan	chyna-bogan
2	Brent Bogan II	brent-bogan-ii
3	Lorenzo Morissette	lorenzo-morissette
4	Mr. Rhiannon Goyette II	mr-rhiannon-goyette-ii
5	Blanche Kling	blanche-kling
6	Terrence Schneider	terrence-schneider
7	Golden Wehner	golden-wehner
8	Webster Hoppe V	webster-hoppe-v
9	Dr. Carolyne Mitchell III	dr-carolyne-mitchell-iii
10	Jaquelin Spencer	jaquelin-spencer

id	film_id	filmable_type	filmable_id
1	1	App\Category	10
2	1	App\Actor	1
3	1	App\Actor	7
4	2	App\Category	6
5	2	App\Category	2
6	2	App\Category	8

La page d'accueil

Dans la page d'accueil pour le moment on a ça :



Une liste des films avec des boutons d'action, et aussi un choix selon la catégorie. Tout ça fonctionne encore mais ça serait bien d'ajouter le choix par acteur aussi.

On avait utilisé un view composer pour générer l'ensemble des catégories pour les vues (dans **AppServiceProvider**) :

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

On va ajouter les acteurs :

```
$view->with('actors', Actor::all());
```

Route

Il nous faut aussi ajouter une route pour la sélection des films par acteur :

```
Route::get('actor/{slug}/films', 'FilmController@index')->name('films.actor');
```

```
GET|HEAD  | actor/{slug}/films     | films.actor          | App\Http\Controllers\FilmController@index          | web
```

Contrôleur

Dans le contrôleur pour l emoment on a ce code :

```
public function index($slug = null)
{
   $query = $slug ? Category::whereSlug($slug)->firstOrFail()->films() : Film::query();
   $films = $query->withTrashed()->oldest('title')->paginate(5);
   return view('index', compact('films', 'slug'));
}
```

On distingue le cas où on a un paramètre, donc un slug et le cas où on n'en a pas. C'était parfait avec juste les catégories mais plus maintenant parce qu'on a deux cas avec des slugs. On va donc le changer ainsi :

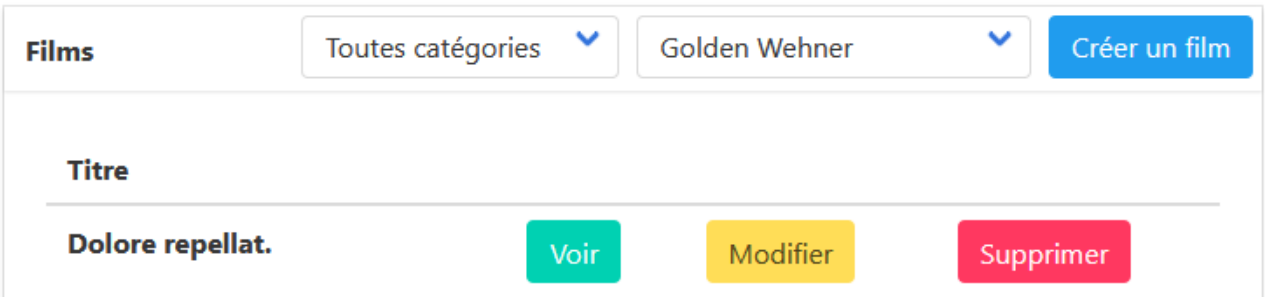
```
use App\{Film, Category, Actor};

...

public function index($slug = null)
{
   $model = null;

   if($slug) {
      if(Route::currentRouteName() == 'films.category') {
         $model = new Category;
      } else {
         $model = new Actor;
      }
   }

   $query = $model ? $model->whereSlug($slug)->firstOrFail()->films() : Film::query();
   $films = $query->withTrashed()->oldest('title')->paginate(5);
   return view('index', compact('films', 'slug'));
}
```

La vue index

Il ne nous reste plus qu'à mettre à jour la vue **index** :

```
<div class="select">
   <select onchange="window.location.href = this.value">
      <option value="{{ route('films.index') }}" @unless($slug) selected @endunless>Tous
acteurs</option>
      @foreach($actors as $actor)
         <option value="{{ route('films.actor', $actor->slug) }}" {{ $slug == $actor->slug ? 'selected'
: '' }}>{{ $actor->name }}</option>
      @endforeach
   </select>
</div>
```

On ajoute la liste à côté de celle des catégories :



Et tout fonctionne ! Mais c'est soit le choix par catégorie, soit pas acteur. On ne peut pas combiner les deux avec notre code mais on va s'en contenter...

Voir un film

Dans la fiche d'un film on n'a pas encore les acteurs, alors on va les ajouter. Si vous vous rappelez on a fait de la liaison implicite pour faire de l'eager loading des catégories quand on transmet l'identifiant d'un film dans l'url (**RouteServiceProvider**) :

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

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

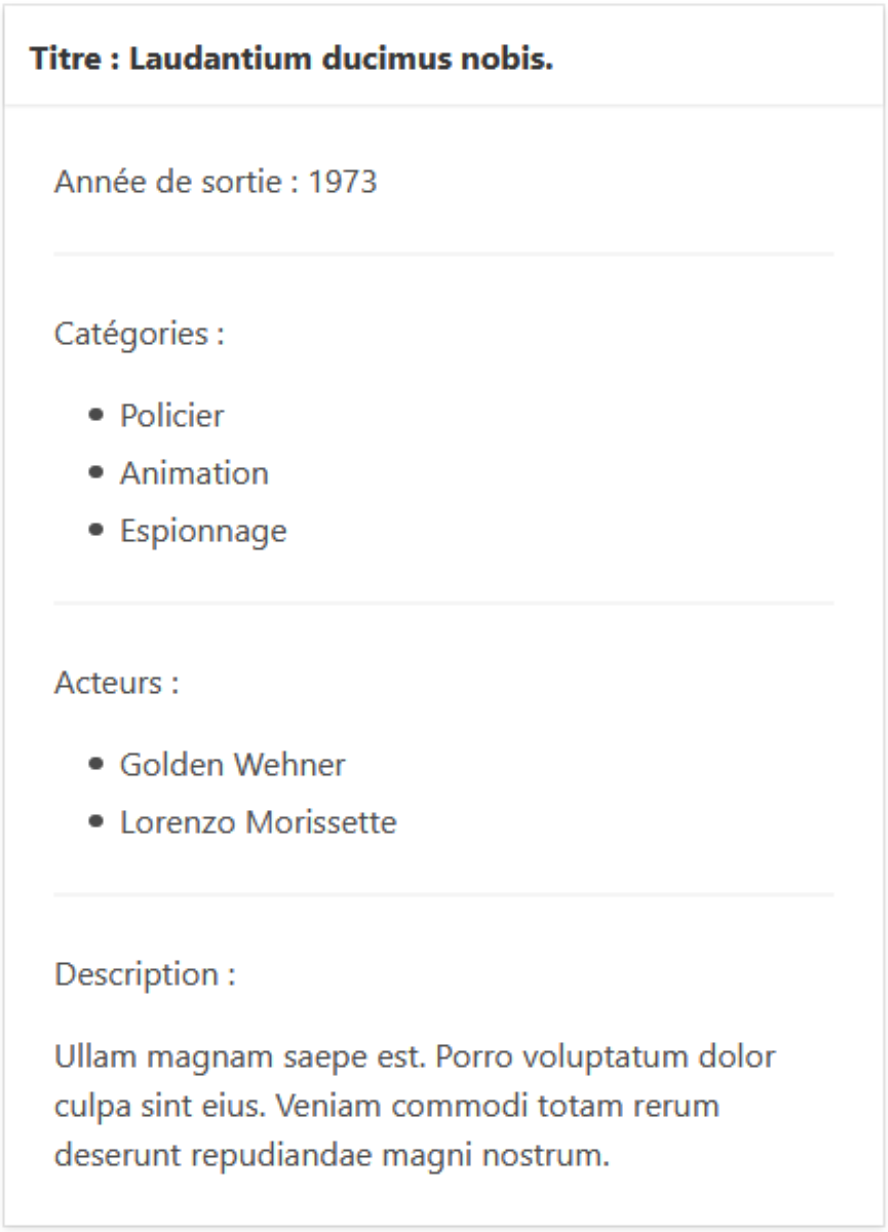
On va donc ajouter les acteurs :

```
return Film::with('categories', 'actors')->find($value) ?? abort(404);
```

Dans la vue **show** on reproduit le code des catégories pour les acteurs :

```
<p>Catégories :</p>
<ul>
    @foreach($film->categories as $category)
        <li>{{ $category->name }}</li>
    @endforeach
</ul>
<hr>
<p>Acteurs :</p>
<ul>
    @foreach($film->actors as $actor)
        <li>{{ $actor->name }}</li>
    @endforeach
</ul>
<hr>
```

Et voilà le résultat :

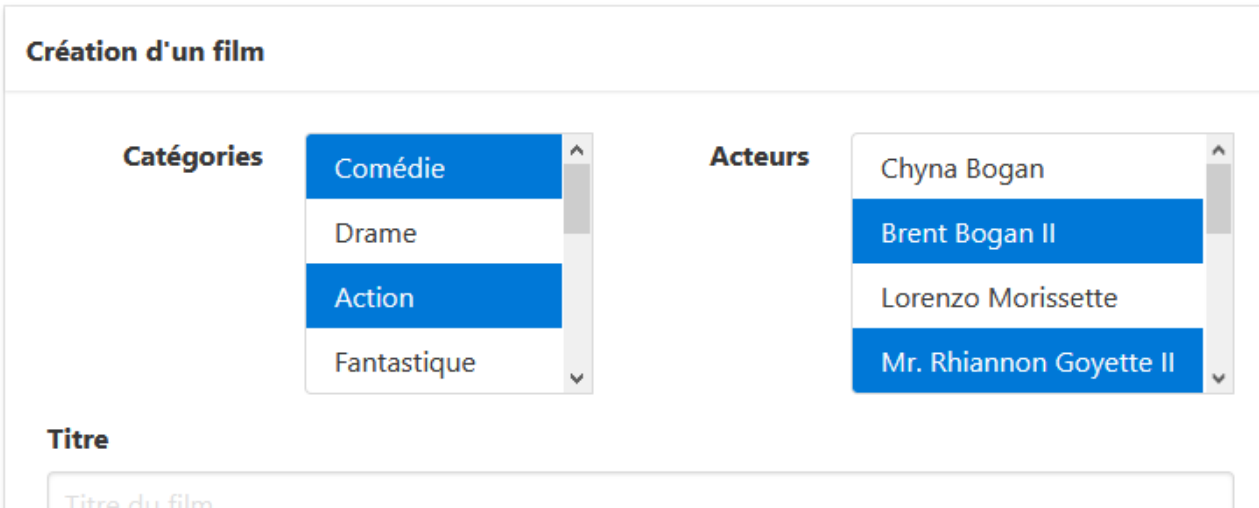


Création d’un film

Dans le formulaire de création d’un film on va devoir ajouter le choix des acteurs. On change la vue **show** :

```
<form action="{{ route('films.store') }}" method="POST">
    @csrf
    <div class="field is-grouped is-horizontal">
        <label class="label field-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' : '' }}>{{ $category->name }}</option>
                @endforeach
            </select>
        </div>
        <label class="label field-label">Acteurs</label>
        <div class="select is-multiple">
            <select name="acts[]" multiple>
                @foreach($actors as $actor)
                    <option value="{{ $actor->id }}" {{ in_array($actor->id, old('acts') ?: []) ? 'selected' : ''
}}>{{ $actor->name }}</option>
                @endforeach
            </select>
        </div>
    </div>
```

On a ainsi les deux listes de choix :



Dans le contrôleur on prévoit l'attachement pour les acteurs en plus des catégories :

```
public function store(FilmRequest $filmRequest)
{
    $film = Film::create($filmRequest->all());

    $film->categories()->attach($filmRequest->cats);
    $film->actors()->attach($filmRequest->acts);

    return redirect()->route('films.index')->with('info', 'Le film a bien été créé');
}
```

Et ça devrait être bon !

Modification d’un film

Dans le formulaire de modification d’un film on va devoir aussi ajouter le choix des acteurs. On change la vue **edit** :

```
<form action="{{ route('films.update', $film->id) }}" method="POST">
   @csrf
   @method('put')
   <div class="field is-grouped is-horizontal">
      <label class="label field-label">Catégories</label>
      <div class="select is-multiple">
         <select name="cats[]" multiple>
            @foreach($categories as $category)
               <option value="{{ $category->id }}" {{ in_array($category->id, old('cats') ?: $film->categories->pluck('id')->toArray()) ? 'selected' : '' }}>{{ $category->name }}</option>
            @endforeach
         </select>
      </div>
      <label class="label field-label">Acteurs</label>
      <div class="select is-multiple">
         <select name="acts[]" multiple>
            @foreach($actors as $actor)
               <option value="{{ $actor->id }}" {{ in_array($actor->id, old('acts') ?: $film->actors->pluck('id')->toArray()) ? 'selected' : '' }}>{{ $actor->name }}</option>
            @endforeach
         </select>
      </div>
   </div>
```

Et dans le contrôleur on doit synchroniser aussi les acteurs :

```
public function update(FilmRequest $filmRequest, Film $film)
{
   $film->update($filmRequest->all());

   $film->categories()->sync($filmRequest->cats);
   $film->actors()->sync($filmRequest->acts);

   return redirect()->route('films.index')->with('info', 'Le film a bien été modifié');
}
```

Et on a terminé le travail ! Avec toutefois un bémol… Quand on supprime un acteur ou une catégorie, étant donné qu’on n’a pas de cascade, la table pivot reste renseignée avec des enregistrements orphelins. Mais c’est une autre histoire qui nous entrainerait un peu trop loin pour le moment. Il y a aussi pas mal de code redondant dans les vues et il y aurait certainement quelque chose à faire aussi de ce côté.

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

En résumé

- Lorsque plusieurs tables sont concernées d’un côté d’une relation on doit appliquer le polymorphisme.
- Laravel propose de nombreuses méthodes pour gérer le polymorphisme selon la situation.