# Cours Laravel 6 – les données – la relation 1:n

laravel.sillo.org/cours-laravel-6-les-donnees-la-relation-1n/

bestmomo

3 septembre 2019

Pour le moment nous n'avons manipulé qu'une table avec Eloquent. Dans le présent chapitre nous allons utiliser deux tables et les mettre en relation.

La relation la plus répandue et la plus simple entre deux tables est celle qui fait correspondre un enregistrement d'une table à plusieurs enregistrements de l'autre table, on parle de relation de un à plusieurs ou encore de relation de type **1:n**.

Nous allons poursuivre notre gestion de films. Comme nous en avons beaucoup nous éprouvons la nécessité de les classer en catégories : comédie, fantastique, drame, thriller...

On va partir du projet dans l'état où on l'a laissé dans le précédent article. Il y a un lien de téléchargement à la fin.

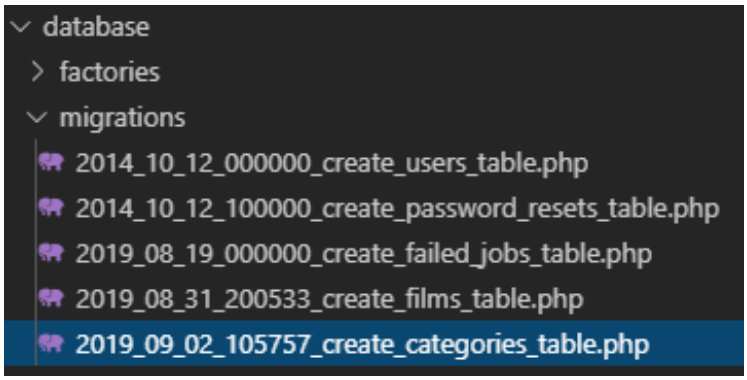
## Les migrations

### La table categories

Nous allons créer une table pour les catégories. On crée sa migration en même temps que le modèle :

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

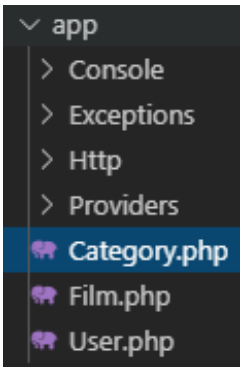
Voici le code complété pour la migration :




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

On va donc définir les colonnes ;

- **name** : nom de la catégorie
- **slug** : adaptation du nom pour le rendre compatible avec les urls




## La table films

On a déjà une migration pour la table **films** mais il va falloir la compléter pour pouvoir la mettre en relation avec la table **categories** :

```
public function up()
{

    Schema::disableForeignKeyConstraints();
    Schema::create('films', function (Blueprint $table) {
        ...
        $table->unsignedBigInteger('category_id');
        $table->foreign('category_id')
            ->references('id')
            ->on('categories')
            ->onDelete('restrict')
            ->onUpdate('restrict');
    });
}
```

Dans la table **films** on déclare une clé étrangère (**foreign**) nommée **category_id** qui référence (**references**) la **colonne id** dans la table (**on**) **categories**. En cas de suppression (**onDelete**) ou de modification (**onUpdate**) on a une restriction (**restrict**).

*Que signifient ces deux dernières conditions ?*

Imaginez que vous avez une catégori avec l'id 5 qui a deux films, donc dans la table **films** on a deux enregistrements avec **category_id** qui a la valeur 5. Si on supprime la catégorie que va-t-il se passer ? On risque de se retrouver avec nos deux enregistrements dans la table **films** avec une clé étrangère qui ne correspond à aucun enregistrement dans la table **categories**. En mettant **restrict** on empêche la suppression d'une catégorie qui a des films. On doit donc commencer par supprimer ses films avant de le supprimer lui-même. On dit que la base assure l'intégrité référentielle. Elle n'acceptera pas non plus qu'on utilise pour **category_id** une valeur qui n'existe pas dans la table **categories**.

Une autre possibilité est **cascade** à la place de **restrict**. Dans ce cas si vous supprimez une catégorie ça supprimera en cascade les films de cette catégorie.

C’est une option qui est rarement utilisée parce qu’elle peut s’avérer dangereuse, surtout dans une base comportant de multiples tables en relation. Mais c’est aussi une stratégie très efficace parce que c’est le moteur de la base de données qui se charge de gérer les enregistrements en relation, vous n’avez ainsi pas à vous en soucier au niveau du code.

On pourrait aussi ne pas signaler à la base qu’il existe une relation et la gérer seulement dans notre code. Mais c’est encore plus dangereux parce que la moindre erreur de gestion des enregistrements dans votre code risque d’avoir des conséquences importantes dans votre base avec de multiples incohérences.

On va lancer les migrations en raffraichissant la base :

```
php artisan migrate:fresh
```

Les migrations sont effectuées dans l’ordre alphabétique, ce qui peut générer un problème avec les clés étrangères. Si la table référencée est créée après on va tomber sur une erreur du genre :

```
Illuminate\Database\QueryException  : SQLSTATE[HY000]: General error: 1215 Cannot add foreign key constraint (SQL: alter table `films` add constraint `films_category_id_foreign` foreign key (`category_id`) references `categories` (`id`) on delete restrict on update restrict)
```

C’est pour cette raison que j’ai ajouté cette ligne dans la migration :

```
Schema::disableForeignKeyConstraints();
```

On désactive temporairement le contrôle référentiel le temps de créer les tables.

## La population

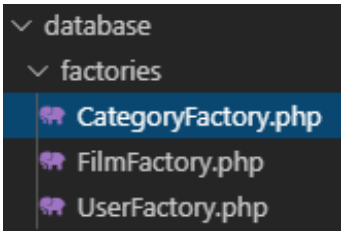
On va remplir les tables avec des enregistrements pour nos essais.

## Les catégories

On va définir un certain nombre de catégories. Alors on crée un factory :

```
php artisan make:factory CategoryFactory --model=Category
```

On complète le code :



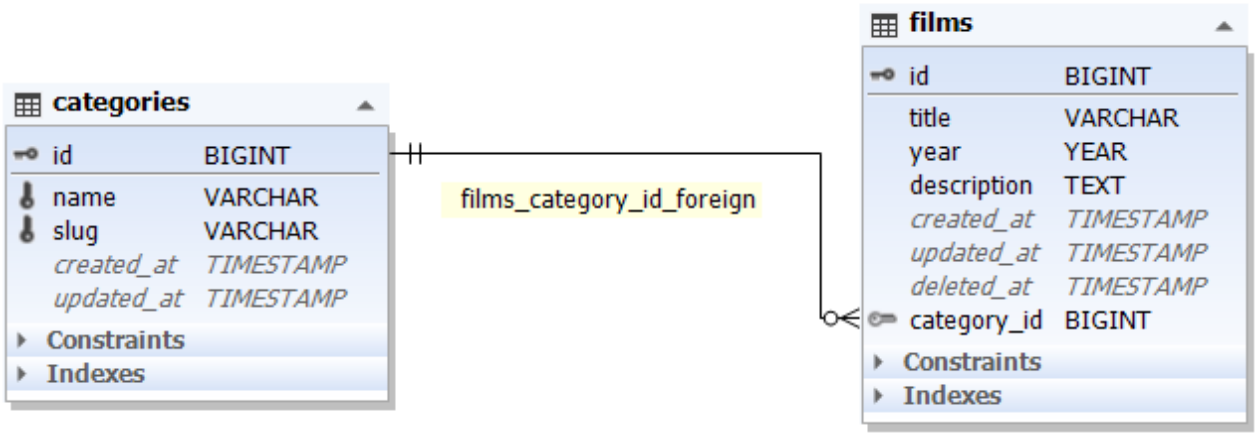
```php
<?php

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

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

## La relation

On a la situation suivante :

- une catégorie peut avoir plusieurs films,
- un film n'appartient qu'à une catégorie.

Il faut trouver un moyen pour référencer cette relation dans les tables. Le principe est simple et on l'a vu dans les migrations : on prévoit dans la table **films** une colonne destinée à recevoir l'identifiant de la catégorie. On appelle cette ligne une **clé étrangère** parce qu'on enregistre ici la clé d'une autre table. Voici une représentation visuelle de cette relation :



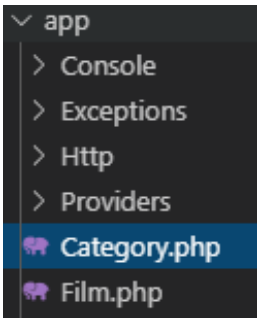

Vous voyez la relation **films_category_id_foreign** dessinée entre la clé **id** dans la table **categories** et la clé étrangère **category_id** dans la table **films**.

## Les modèles et la relation

### Le modèle Category

Le modèle **Category** se trouve ici (on l'a créé en même temps que la migration) :

On va lui ajouter ce code :




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

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

## Le modèle Film

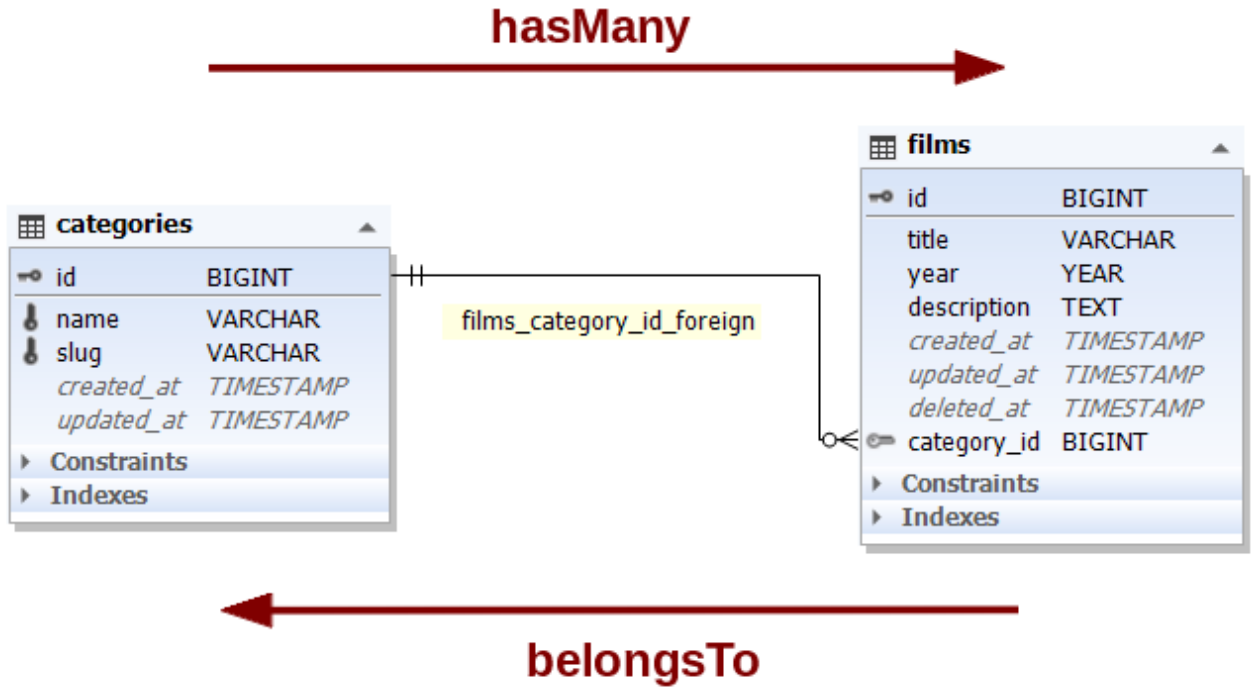
Dans le modèle Film on va coder la réciproque :

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

Ici on a la méthode **category** (au singulier) qui permet de trouver la catégorie à laquelle appartient (**belongsTo**) le film.

## La relation 1:n

Voici une schématisation de la relation avec les deux méthodes :




*Si vous ne spécifiez pas de manière explicite le nom de la table dans un modèle, Laravel le déduit à partir du nom du modèle en le mettant au pluriel (à la mode anglaise) et en mettant la première lettre en minuscule. Donc avec le modèle **Film** il en conclut que la table s'appelle **films**. Si ce n'était pas satisfaisant il faudrait créer une propriété **$table**.*

Les deux méthodes mises en place permettent de récupérer facilement un enregistrement lié. Par exemple pour avoir tous les films de la catégorie qui a l'id 1 :

```
$films = App\Models\Category::find(1)->films;
```

De la même manière on peut trouver la catégorie du film d'id 1 :

```
$category = App\Models\Film::find(1)->category;
```

Vous voyez que le codage devient limpide avec ces méthodes !

## La population (seeding)

Pour la population on va justement utiliser la relation qu'on vient de mettre en place. On modifie ainsi **DatabaseSeeder** :

```
public function run()
{
   factory(App\Category::class, 10)->create()->each(function ($category) {
      $i = rand(2, 8);
      while (--$i) {
         $category->films()->save(factory(App\Film::class)->make());
      }
   });
}
```

On lance la population :

```
php artisan db:seed
```

On crée ainsi 10 catégories :

Avec cette population pour les catégories le nom et le slug sont indentiques mais dans une situation plus réaliste il y aurait évidemment souvent des différences, avec les majuscules, les espaces, les accents…

Et pour chacune entre 1 et 7 films associés.

On a maintenant tout en place pour commencer à nous amuser…

| id | name | slug |
|---|---|---|
| 1 | a | a |
| 2 | dignissimos | dignissimos |
| 3 | aut | aut |
| 4 | est | est |
| 5 | ratione | ratione |
| 6 | sint | sint |
| 7 | voluptatum | voluptatum |
| 8 | pariatur | pariatur |
| 9 | maiores | maiores |
| 10 | voluptates | voluptates |

| id | category_id | title |
|---|---|---|
| 1 | 1 | Consectetur impedit. |
| 2 | 1 | Excepturi voluptatibus. |
| 3 | 1 | Vero necessitatibus rerum. |
| 4 | 2 | Nihil beatae. |
| 5 | 2 | Nisi et qui. |
| 6 | 3 | Quaerat ullam quos. |
| 7 | 4 | Ex animi et. |
| 8 | 4 | Voluptatem quia. |
| 9 | 4 | Omnis aut. |
| 10 | 4 | Consequatur repudiandae. |

## Route et contrôleur

On va définir une nouvelle route pour aller chercher les films par catégorie :

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

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

On voit qu'on pointe la méthode **index** du contrôleur. Cette méthode existe déjà mais ne servait jusque là qu'à fournir la liste paginée des films sans tenir compte de catégories. On met à jour le code :

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

...

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

On voit qu'on distingue le cas où on fournit un slug de catégorie et le cas où on n'en fournit pas. On envoie toutes les informations nécessaires dans la vue **index**.

## La vue index

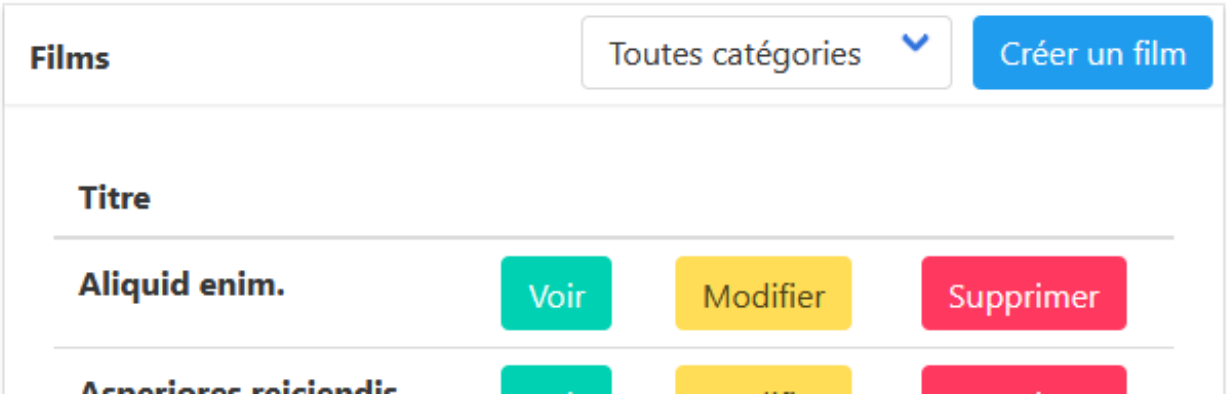
On va donc un peu modifier la vue **index** pour ajouter cette fonctionnalité. Essentiellement on ajoute une liste de choix :

```
<div class="card">
   <header class="card-header">
      <p class="card-header-title">Films</p>
      <div class="select">
         <select onchange="window.location.href = this.value">
            <option value="{{ route('films.index') }}" @unless($slug) selected @endunless>Toutes
catégories</option>
            @foreach($categories as $category)
               <option value="{{ route('films.category', $category->slug) }}" {{ $slug == $category-
>slug ? 'selected' : '' }}>{{ $category->name }}</option>
            @endforeach
         </select>
      </div>
      <a class="button is-info" href="{{ route('films.create') }}">Créer un film</a>
   </header>
```

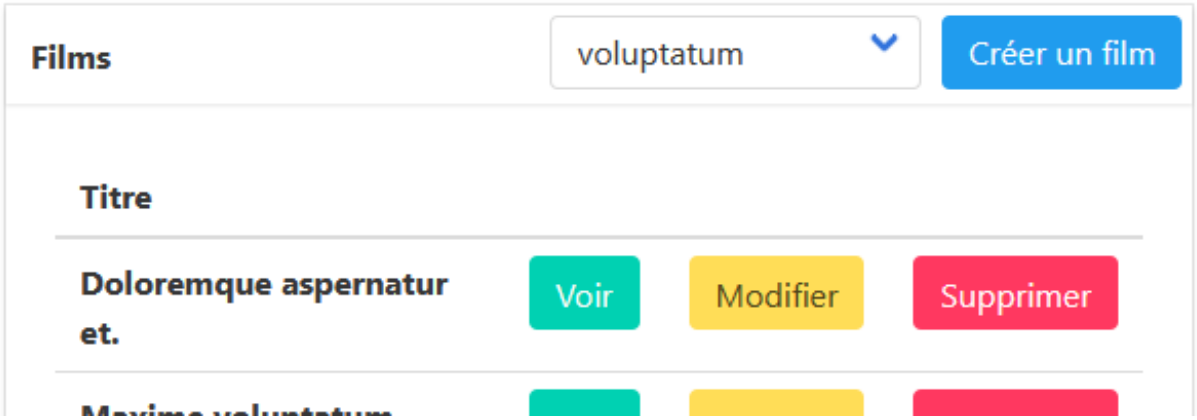
On modifie aussi un peu le style pour tenir compte de la liste :

```
select, .is-info {
   margin: 0.3em;
}
```

Maintenant quand on arrive avec l'url **.../films** on a :




Quand on clique sur une option de la liste, donc une des catégories, ça envoie la requête qui va bien et ça affiche les films de cette catégorie :




Ici on a l'url **.../category/voluptatum/films**.

## La vue show

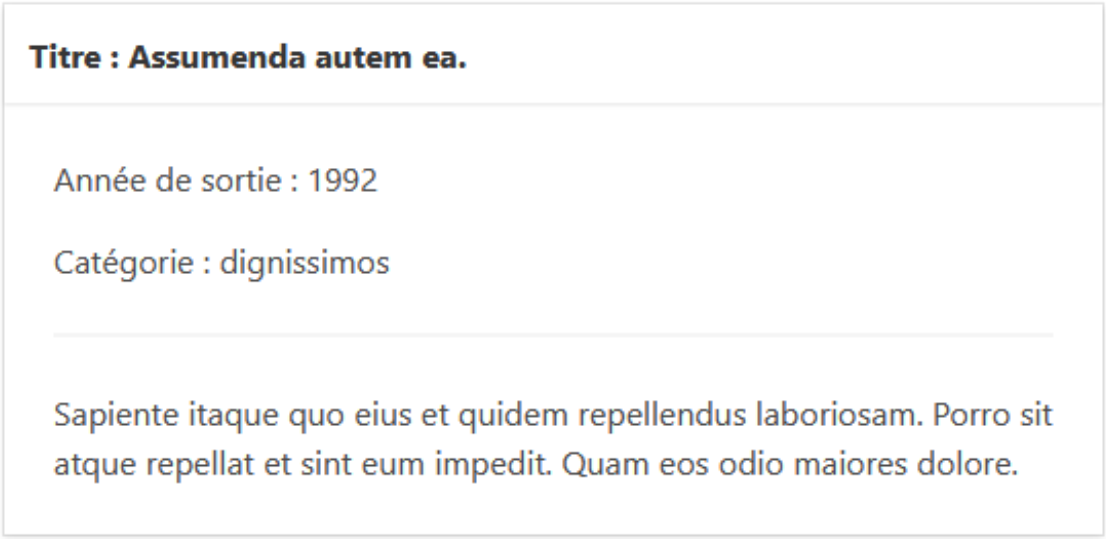
On va aussi ajouter dans la vue **show** le nom de la catégorie du film. On complète le contrôleur :

```
public function show(Film $film)
{
   $category = $film->category->name;
   return view('show', compact('film', 'category'));
}
```

Et la vue :

```
<div class="content">
   <p>Année de sortie : {{ $film->year }}</p>
   <p>Catégorie : {{ $category }}</p>
   ...
</div>
```

Et ça roule :



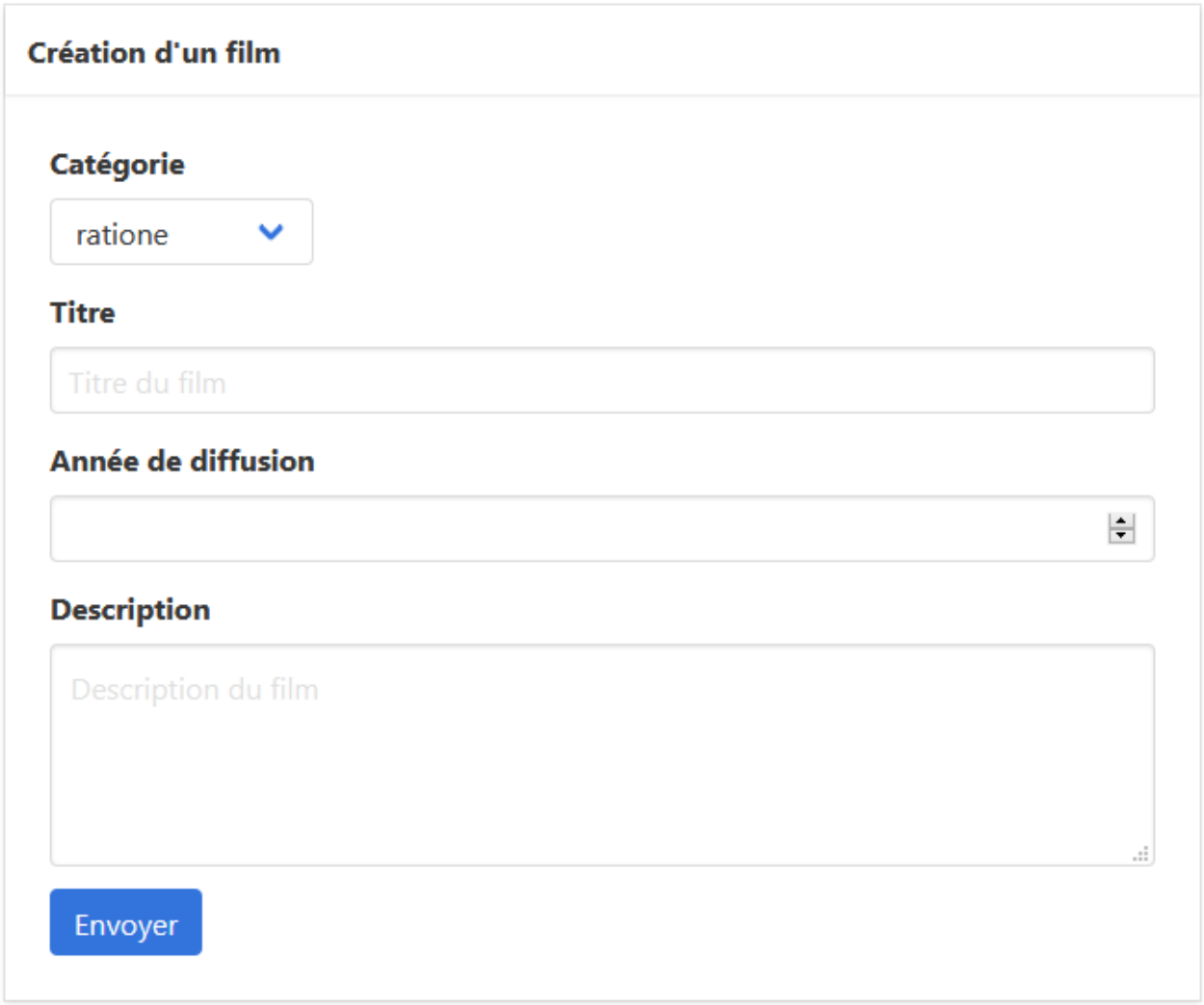

## La création d'un film

Maintenant aussi quand on crée un film il faut l'attribuer à une catégorie. Il faut déjà modifier la méthode **create** du contrôleur pour envoyer les catégories :

```
public function create()
{
   $categories = Category::all();
   return view('create', compact('categories'));
}
```

Et modifier la vue **create** en ajoutant une liste des catégories :

```
<form action="{{ route('films.store') }}" method="POST">
  @csrf
  <div class="field">
    <label class="label">Catégorie</label>
    <div class="select">
      <select name="category_id">
        @foreach($categories as $category)
          <option value="{{ $category->id }}">{{ $category->name }}</option>
        @endforeach
      </select>
    </div>
  </div>
```




On ajoute le champ **category_id** dans la propriété **$fillable** du modèle **Film** :

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

Et on n'a même pas besoin de toucher à la méthode **store** du contrôleur ! D'autre part on ne va pas se soucier de validation pour une liste imposée.

## Les composeurs de vue

Vous avez peut-être remarqué une répétition de code dans ces deux méthodes du contrôleur :

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

public function create()
{
   $categories = Category::all();
   return view('create', compact('categories'));
}
```

Dans les deux cas on va chercher toutes les catégories pour les envoyer à la vue. Laravel propose le concept de composeur de vue (view composer) pour traiter cette situation de façon plus élégante. Dans un premier temps on nettoie le contrôleur :

```
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'));
}

public function create()
{
   return view('create');
}
```

Donc là on envoie plus les catégories et ça ne fonctionne plus !

Changez ainsi le code du fichier **App\Providers\AppServiceProvider** :

```php
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\View;
use App\Category;

class AppServiceProvider extends ServiceProvider
{
   public function register()
   {
      //
   }

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

On utilise la façade **View** avec la méthode **composer** pour mettre en place le fait que chaque fois qu'une des deux vues **index** ou **create** et appelée alors on associe la variable **categories** qui contient toutes les catégories. Et ça fonctionne ! Classiquement on créerait plutôt un provider dédié à cette tâche.

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

## En résumé

- Une relation de type **1:n** nécessite la création d'une clé étrangère côté **n**.
- Une relation dans la base nécessite la mise en place de méthodes spéciales dans les modèles.
- On dispose de plusieurs méthodes pour manipuler une relation.
- Un composeur de vue permet de mutualiser du code pour envoyer des informations dans les vues.