

# Cours Laravel 6 – les données – les ressources (1/2)

 [laravel.sillo.org/cours-laravel-6-les-donnees-les-ressources/](https://laravel.sillo.org/cours-laravel-6-les-donnees-les-ressources/)

bestmomo

September 1,  
2019

Dans ce chapitre nous allons commencer à étudier les ressources qui permettent de créer des routes « CRUD » (**C**reate, **R**ead, **U**ppdate, **D**eleter) adaptées à la persistance de données. Comme exemple pratique nous allons prendre le cas d'une table de films.

## Les données

On repart d'un Laravel vierge et on crée une base comme on l'a vu précédemment. Appelons la par exemple **laravel6** pour faire original. On renseigne le fichier **.env** en conséquence :

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel6
DB_USERNAME=root
DB_PASSWORD=
```

## La migration

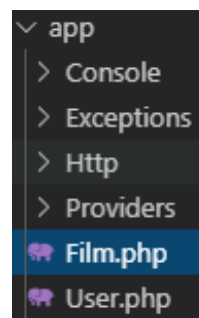
On va créer avec Artisan le modèle **Film** en même temps que la migration :

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

```
λ php artisan make:model Film --migration
Model created successfully.
Created Migration: 2019_08_31_200533_create_films_table
```

Pour faire simple on va se contenter de 3 colonnes pour le titre du film, son année de sortie et sa description :

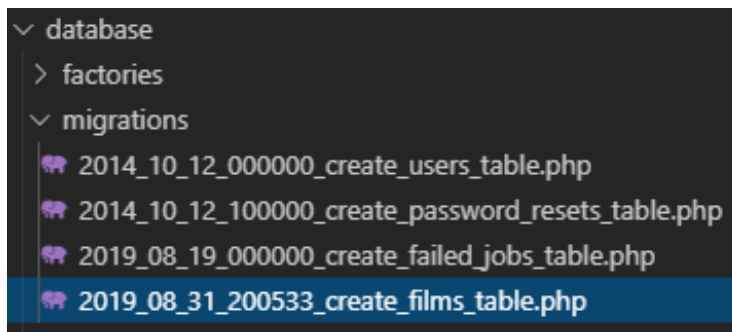
```
public function up()
{
    Schema::create('films', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->string('title');
        $table->year('year');
        $table->text('description');
        $table->timestamps();
    });
}
```



On a les champs :

- **id** : entier auto-incrémenté qui sera la clé primaire de la table,

- **name** : texte pour le nom,
- **title** : texte pour le nom du film,
- **year** : année de sortie du film,
- **description** : description du film,
- **created\_at** et **updated\_at** créés par la méthode **timestamps**,



Ensuite on lance la migration :

```
λ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.77 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.67 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.43 seconds)
Migrating: 2019_08_31_200533_create_films_table
Migrated: 2019_08_31_200533_create_films_table (0.34 seconds)
```

On a la création des tables de base de Laravel qu'on a déjà vues et qui ne nous intéressent pas pour cet article. Mais on voit aussi la création de la table **films** :

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	<b>id</b>	bigint(20)		UNSIGNED	Non	Aucun(e)		AUTO_INCREMENT
2	<b>title</b>	varchar(255)	utf8mb4_unicode_ci		Non	Aucun(e)		
3	<b>year</b>	year(4)			Non	Aucun(e)		
4	<b>description</b>	text	utf8mb4_unicode_ci		Non	Aucun(e)		
5	<b>created_at</b>	timestamp			Oui	NULL		
6	<b>updated_at</b>	timestamp			Oui	NULL		

## Le modèle

Le modèle **Film** qu'on a créé est vide au départ. On va se contenter de prévoir l'assignement de masse avec la propriété **\$fillable** :

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

## La population

Pour nos essais on va remplir un peu la table avec quelques films. On va créer un factory :

```
php artisan make:factory FilmFactory -mApp\Film
```

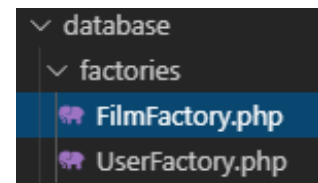
Comme on a déclaré le modèle le code est déjà bien avancé :

```
<?php

/** @var \Illuminate\Database\Eloquent\Factory $factory */

use App\Film;
use Faker\Generator as Faker;

$factory->define(Film::class, function (Faker $faker) {
    return [
        //
    ];
});
```



On va compléter ainsi :

```
$factory->define(Film::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence(2, true),
        'year' => $faker->year,
        'description' => $faker->paragraph(),
    ];
});
```

On change ensuite le code du seeder (**database/seeds/DatabaseSeeder.php**) :

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

Il ne reste plus qu'à lancer la population :

```
php artisan db:seed
```

Si tout va bien on se retrouve avec 10 films dans la table :

id	title	year	description
1	Aspernatur quo illo.	1981	Ut tempora nobis et dolore. Ut et commodi laudant...
2	Officia illum eos.	1996	Sit adipisci ullam beatae. Sed rerum omnis quos vo...
3	Doloremque architecto.	1978	Quo aut dolores quibusdam quo expedita id. Hic exp...
4	Ea est.	1982	Aut voluptatum et officia voluptatem. Labore elige...
5	Consequuntur voluptatem modi.	1985	Recusandae assumenda consequuntur commodi adipisci...
6	Voluptates enim.	2014	Nesciunt ab quaerat vel aperiam voluptas ut aut no...
7	Porro in.	1973	Vel molestias illo perspiciatis eos fuga ut ut. Nu...
8	Tempora non.	1988	Architecto qui et debitis eligendi mollitia ducimu...
9	Repellat et.	1986	Rerum neque earum atque ad molestiae soluta. Corpo...
10	Provident qui.	2014	Quis dolore et a id. Ad velit enim repellendus est...

# Une ressource

---

## Le contrôleur

---

On va maintenant créer un contrôleur de ressource avec Artisan :

```
php artisan make:controller FilmController --resource
```

C'est la commande qu'on a déjà vue pour créer un contrôleur avec en plus l'option - **resource**.

Vous trouvez comme résultat le contrôleur **app/Http/Controllers/FilmController** :

Avec ce code :

```
<?php

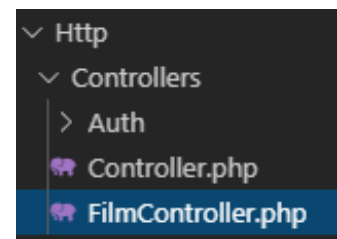
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class FilmController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }
}
```



```

}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    //
}

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    //
}

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    //
}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id)
{
    //
}
}

```

Les 7 méthodes créées couvrent la gestion complète des films:

- **index** : pour afficher la liste des films,
- **create** : pour envoyer le formulaire pour la création d'un nouveau film,

- **store** : pour créer un nouveau film,
- **show** : pour afficher les données d'un film,
- **edit** : pour envoyer le formulaire pour la modification d'un film,
- **update** : pour modifier les données d'un film,
- **destroy** : pour supprimer un film.

## Les routes

Pour créer toutes les routes il suffit de cette unique ligne de code :

```
Route::resource('films', 'FilmController');
```

On va vérifier ces routes avec Artisan :

Method	URI	Name	Action	Middleware
GET HEAD	/		Closure	web
GET HEAD	films	films.index	App\Http\Controllers\FilmController@index	web
POST	films	films.store	App\Http\Controllers\FilmController@store	web
GET HEAD	films/create	films.create	App\Http\Controllers\FilmController@create	web
GET HEAD	films/{film}	films.show	App\Http\Controllers\FilmController@show	web
PUT PATCH	films/{film}	films.update	App\Http\Controllers\FilmController@update	web
DELETE	films/{film}	films.destroy	App\Http\Controllers\FilmController@destroy	web
GET HEAD	films/{film}/edit	films.edit	App\Http\Controllers\FilmController@edit	web

Vous trouvez 7 routes, avec chacune une méthode et une url, qui pointent sur les 7 méthodes du contrôleur. Notez également que chaque route a aussi un nom qui peut être utilisé par exemple pour une redirection. On retrouve aussi pour chaque route le middleware **web** dont je vous ai déjà parlé.

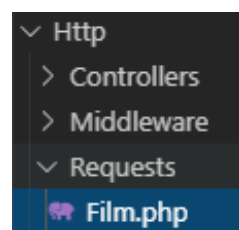
Nous allons à présent considérer chacune de ces routes et créer la gestion des données, les vues, et le code nécessaire au niveau du contrôleur.

## La validation

On va créer une requête de formulaire pour la création ou la modification d'un film :

```
php artisan make:request Film
```

On a deux champs à vérifier : le titre et l'année. A priori il n'y a aucune différence de validation pour la création et la modification. Voilà le code modifié pour cette classe :



```

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class Film extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'title' => ['required', 'string', 'max:100'],
            'year' => ['required', 'numeric', 'min:1950', 'max:' . date('Y')],
            'description' => ['required', 'string', 'max:500'],
        ];
    }
}

```

On prévoi comme règles :

- **title** : le champ est obligatoire (**required**), ça doit être du texte (**string**), le nombre maximum de caractères (**max**) doit être de 100
- **year** : le champ est obligatoire (**required**), ça doit être un nombre (**number**), la valeur minimale (**min**) doit être 1950, la valeur maximale (**max**) doit être l'année actuelle (date('Y'))
- **description** : le champ est obligatoire (**required**), ça doit être du texte (**string**), le nombre maximum de caractères (**max**) doit être de 500

On pourra injecter cette classe dans les méthodes du contrôleur.

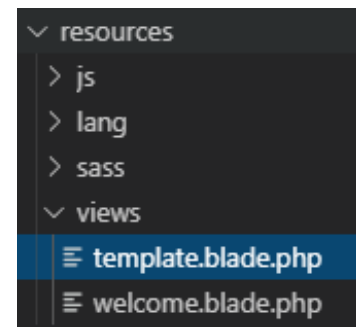
## Le template

Laravel s'occupe essentiellement du côté serveur et n'impose rien côté client, même s'il propose des choses. Autrement dit on peut utiliser Laravel avec n'importe quel système côté client. Pour notre exemple je vous propose d'utiliser Bulma pour la mise en forme. Voici un template qui va nous servir pour toutes nos vues :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <title>Films</title>
    <link rel="stylesheet"

```



```

href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.7.5/css/bulma.min.css">
  @yield('css')
</head>
<body>
  <main class="section">
    <div class="container">
      @yield('content')
    </div>
  </main>
</body>
</html>

```

## La liste des films

---

### La route

---

La liste des films correspond à cette route :

```
GET|HEAD | films | films.index | App\Http\Controllers\FilmController@index | web
```

### Le contrôleur

---

Dans le contrôleur c'est la méthode **index** qui est concernée. On va donc la coder :

```

...
use App\Film;

class FilmController extends Controller
{
  public function index()
  {
    $films = Film::all();
    return view('index', compact('films'));
  }
}
...

```

On va chercher tous les films avec la méthode **all** du modèle, on appelle la vue **index** en lui transmettant les films.

### La vue index

---



On crée la vue **index** :

Avec ce code :

```
@extends('template')
```

```
@section('content')
```

```
<div class="card">
```

```
<header class="card-header">
```

```
<p class="card-header-title">Films</p>
```

```
</header>
```

```
<div class="card-content">
```

```
<div class="content">
```

```
<table class="table is-hoverable">
```

```
<thead>
```

```
<tr>
```

```
<th>#</th>
```

```
<th>Titre</th>
```

```
<th></th>
```

```
<th></th>
```

```
<th></th>
```

```
</tr>
```

```
</thead>
```

```
<tbody>
```

```
@foreach($films as $film)
```

```
<tr>
```

```
<td>{{ $film->id }}</td>
```

```
<td><strong>{{ $film->title }}</strong></td>
```

```
<td><a class="button is-primary" href="{{ route('films.show', $film->id)
```

```
}}">Voir</a></td>
```

```
<td><a class="button is-warning" href="{{ route('films.edit', $film->id)
```

```
}}">Modifier</a></td>
```

```
<td>
```

```
<form action="{{ route('films.destroy', $film->id) }}" method="post">
```

```
@csrf
```

```
@method('DELETE')
```

```
<button class="button is-danger" type="submit">Supprimer</button>
```

```
</form>
```

```
</td>
```

```
</tr>
```

```
@endforeach
```

```
</tbody>
```

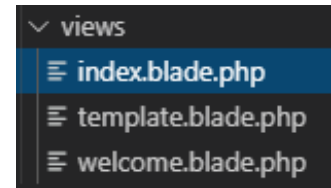
```
</table>
```

```
</div>
```

```
</div>
```

```
</div>
```

```
@endsection
```



Films				
#	Titre			
1	Aspernatur quo illo.	Voir	Modifier	Supprimer
2	Officia illum eos.	Voir	Modifier	Supprimer
3	Doloremque architecto.	Voir	Modifier	Supprimer
4	Ea est.	Voir	Modifier	Supprimer
5	Consequuntur voluptatem modi.	Voir	Modifier	Supprimer
6	Voluptates enim.	Voir	Modifier	Supprimer
7	Porro in.	Voir	Modifier	Supprimer
8	Tempora non.	Voir	Modifier	Supprimer
9	Repellat et.	Voir	Modifier	Supprimer
10	Provident qui.	Voir	Modifier	Supprimer

Quelques remarques concernant le code :

- on utilise la directive **@foreach** pour faire une boucle sur tous les films
- la méthode **route** qui génère une url selon la route peut être accompagnée d'un paramètre, par exemple **route('films.show', \$film->id)** permet de générer l'url de la forme **.../films/id**
- les formulaire HTML ne supportent pas les verbes PUT, PATCH et DELETE, du coup on doit utiliser le verbe POST et prévoir dans le formulaire un input caché qui indique le verbe à utiliser en réalité, la directive **@method** permet de facilement mettre ça en place, on s'en sert pour le bouton de suppression

## La pagination

Ici on n'a que 10 films mais imaginez qu'on en ait des centaines ou des milliers ! Dans ce cas une pagination serait la bienvenue. Laravel est bien équipé pour ça. Au niveau du contrôleur le changement est facile :

```
$films = Film::paginate(5);
```

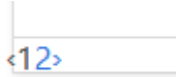
On a remplacé la méthode **all** par **paginate** en indiquant en paramètre le nombre d'enregistrement par page. Ensuite dans la vue il suffit de prévoir ce code :

```

</div>
<footer class="card-footer">
    {{ $films->links() }}
</footer>
</div>
@endsection

```

Mais le souci c'est que par défaut le marquage généré est celui qui convient à Bootstrap 4, du coup avec Bulma on obtient ça :

 Ce qui n'est pas du meilleur effet !

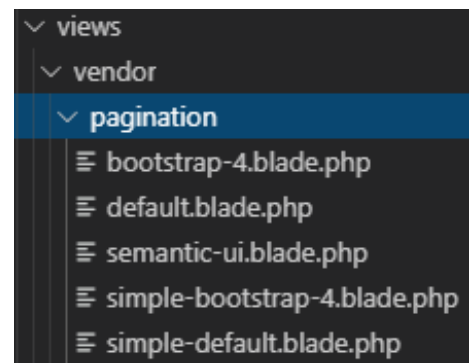
Il nous faut donc modifier la vue qui génère ce code. Comme nous ne sommes pas les premiers à avoir ce souci il suffit de chercher sur la toile et on trouve facilement [ce repo](#).

On commence par publier les vues :

```
php artisan vendor:publish --tag=laravel-pagination
```

On se rend compte d'ailleurs qu'il y en a une de prévue pour Semantic qui est aussi un superbe framework.

Pour terminer on remplace le code du fichier **bootstrap-4.blade.php** par celui-ci :



```

@if ($paginator->hasPages())
    <nav class="pagination is-centered" role="navigation" aria-label="pagination">
        { {-- Previous Page Link --} }
        @if ($paginator->onFirstPage())
            <a class="pagination-previous" disabled>@lang('pagination.previous')</a>
        @else
            <a class="pagination-previous" href="{{ $paginator->previousPageUrl()
        }}">@lang('pagination.previous')</a>
        @endif

        { {-- Next Page Link --} }
        @if ($paginator->hasMorePages())
            <a class="pagination-next" href="{{ $paginator->nextPageUrl()
        }}">@lang('pagination.next')</a>
        @else
            <a class="pagination-next" disabled>@lang('pagination.next')</a>
        @endif

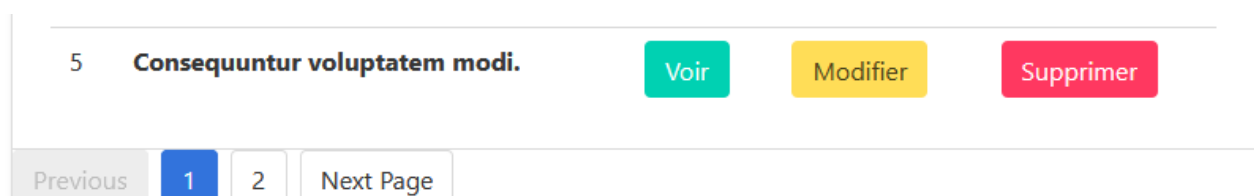
        { {-- Pagination Elements --} }
        <ul class="pagination-list">
            @foreach ($elements as $element)
                { {-- "Three Dots" Separator --} }
                @if (is_string($element))
                    <li><span class="pagination-ellipsis">&hellip;</span></li>
                @endif

                { {-- Array Of Links --} }
                @if (is_array($element))
                    @foreach ($element as $page => $url)
                        @if ($page == $paginator->currentPage())
                            <li><a class="pagination-link is-current" aria-label="Goto page {{ $page }}">
                        {{ $page }}</a></li>
                        @else
                            <li><a href="{{ $url }}" class="pagination-link" aria-label="Goto page {{
                        $page }}">{{ $page }}</a></li>
                        @endif
                    @endforeach
                @endif
            @endforeach
        </ul>

    </nav>
@endif

```

Maintenant c'est un peu mieux :



Ça mérite juste d'être un peu aéré et centré. Comme Bulma utilise Flex on va ajouter

quelques règles dans la vue **index** :

```
@section('css')
<style>
  .card-footer {
    justify-content: center;
    align-items: center;
    padding: 0.4em;
  }
</style>
@endsection
```

Maintenant c'est plus joli :

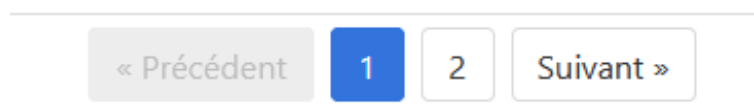
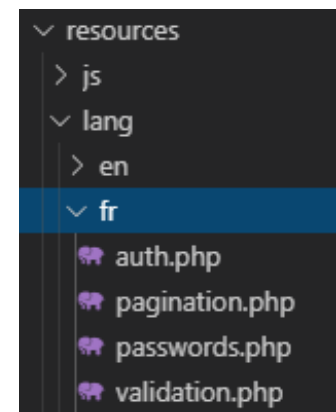


Mais ça serait encore mieux en français ! On a déjà vu qu'on peut récupérer les fichiers de langue ici. On copie le dossier ici :

Et dans **config/app.php** on change cette ligne :

'locale' => 'fr',

Maintenant c'est parfait !



## L'affichage d'un film

On va voir maintenant l'affichage des données d'un film. On y accède à partir du bouton **Voir**.

## La route

La liste des films correspond à cette route :

```
GET HEAD | films/{film} | films.show | App\Http\Controllers\FilmController@show | web
```

## Le contrôleur

---

Dans le contrôleur c'est la méthode **show** qui est concernée. On va donc la coder.

Il me faut toutefois préciser déjà un point important. Dans la version du contrôleur générée par défaut on voit que le film au niveau des arguments des fonctions est référencé par son identifiant, par exemple :

```
public function show($id)
```

La variable **id** contient la valeur passée dans l'url. Par exemple **.../films/8** indique qu'on veut voir les informations du film d'identifiant 8. Il suffit donc ensuite d'aller chercher dans la base le film correspondant.

On va utiliser une autre stratégie :

```
public function show(Film $film)
```

L'argument cette fois est une instance du modèle **App\Film**. Etant donné qu'il rencontre ce type Laravel va automatiquement livrer une instance du modèle pour le film concerné ! C'est ce qu'on appelle liaison implicite (Implicit Binding). Vous voyez encore là à quel point Laravel nous simplifie la vie.

Du coup la méthode devient très simple à coder :

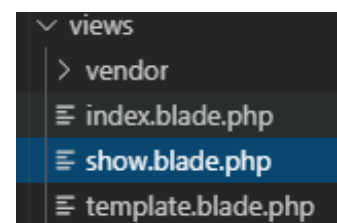
```
public function show(Film $film)
{
    return view('show', compact('film'));
}
```

## La vue show

---

On crée cette vue :

Avec ce code :



```

@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>{{ $film->description }}</p>
            </div>
        </div>
    </div>
@endsection

```

**Titre : Aspernatur quo illo.**

Année de sortie : 1981

Ut tempora nobis et dolore. Ut et commodi laudantium atque reprehenderit aut ab. Et perferendis ipsum enim non modi aperiam consectetur.

Sobre et efficace. J'aurais pu prévoir un bouton de retour mais les navigateurs font déjà ça très bien.

## Supprimer un film

### La route

La suppression d'un film correspond à cette route :

```
DELETE | films/{film} | films.destroy | App\Http\Controllers\FilmController@destroy | web
```

### Le contrôleur

Dans le contrôleur c'est la méthode **destroy** qui est concernée. On va donc la coder :

```
public function destroy(Film $film)
{
    $film->delete();

    return back()->with('info', 'Le film a bien été supprimé dans la base de données.');
```

Comme pour la méthode show on utilise une liaison implicite et on obtient du coup immédiatement une instance du modèle. Comme c'est un peu brutal comme suppression il peut être judicieux de fournir un message de confirmation pour l'utilisateur. Je ne vais pas le faire ici pour ne pas trop alourdir le projet et il s'agit uniquement de traitement côté client.

Par contre après la suppression il faut afficher quelque chose pour dire que l'opération s'est réalisée correctement. On voit qu'il y a une redirection avec la méthode **back** qui renvoie la même page. D'autre part la méthode with permet de flasher une information dans la session. Cette information ne sera valide que pour la requête suivante. Dans notre vue **index** on va prévoir quelque chose pour afficher cette information :

```
@section('content')
    @if(session()->has('info'))
        <div class="notification is-success">
            {{ session('info') }}
        </div>
    @endif
    <div class="card">
```

La directive **@if** permet de déterminer si une information est présente en session, et si c'est le cas de l'afficher :

Le film a bien été supprimé dans la base de données.

Films

Classiquement on prévoit un bouton pour fermer la barre de notification. Là encore je vais simplifier parce c'est aussi un traitement purement client. Il suffit d'ajouter un peu de Javascript pour le faire (vous pouvez consulter [la documentation de Bulma sur le sujet](#)).

Dans le prochain article on verra comment créer et modifier un film.

## En résumé

- Une ressource dans Laravel est constituée d'un contrôleur comportant les 7



méthodes permettant une gestion complète.

- Les routes vers une ressource sont créées avec une simple ligne de code.
- Laravel permet de mettre en place facilement une pagination, il faut adapter l'apparence en fonction du framework CSS qu'on utilise.
- On peut mettre en place dans le routage une liaison implicite pour générer automatiquement une instance de la classe dont l'identifiant est passée dans l'url.