Laravel

Projet table et opérations de recherche, tri et pagination.

Auteur : ***Berthe Seydou***

08/07/2016

Table des matières

[Table et opérations de recherche, tri et pagination : 2](#_Toc458514657)

[1. Création de table : 2](#_Toc458514658)

[2. Interaction avec la base de données (Opération de recherche, tri et pagination) : 4](#_Toc458514659)

[3. Validation des données à insertion dans une table (Intégrité des données): 13](#_Toc458514660)

Prérequis : Projet Inscription Laravel.

# Table et opérations de recherche, tri et pagination :

Dans ce projet, il sera question de voir un peu comment interagir avec la base de données. Pour ce faire, on va répondre aux questions suivantes :

🡪 Comment créer une table ?

🡪 Comment ajouter des champs à une table après création de la table ?

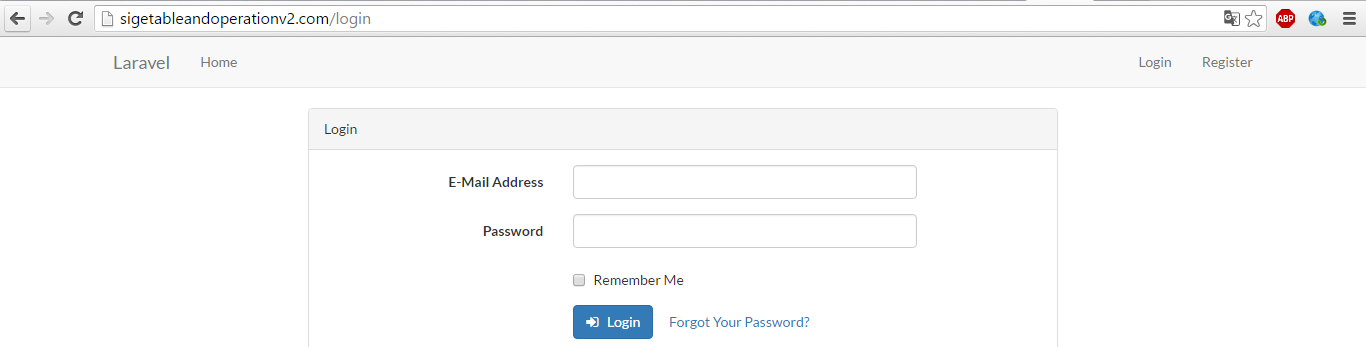
🡪 Comment faire des opérations sur la table dans notre cas (Insertion, listé, Rechercher, trié et la pagination).

🡪Importer une ressource dans la base de données (Seeder).

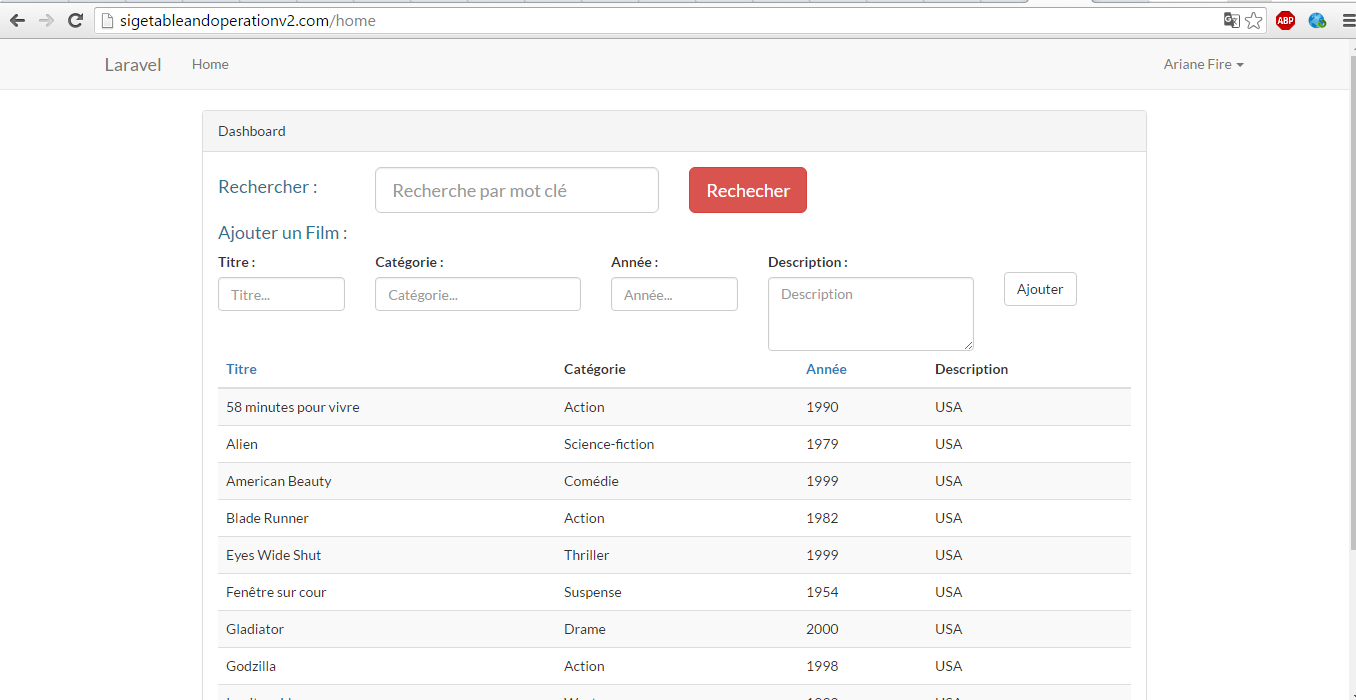
🡪 Comment faire une validation avant d’insérer un élément dans la base de données.

La réponse à ses questions se ferait à travers une application de « gestion basique de Film », en effet il sera question pour un utilisateur de :

🡪S’authentifier :



🡪Ajouter, rechercher, trier et naviguer entre la liste des vidéos :



Commençons tout d’abord à partir d’un projet gérant déjà l’authentification « **AuthRapide** » qui peut être récupérer depuis GIT [AuthRapide](https://github.com/ArianeExtern/EtudeLaravel.git).

Pour les opérations, nous allons mettre en place un gestionnaire très basique de films où on peut s’authentifier, ajouter des films, trier les films et faire la pagination des films.

Avant d’entamer la première partie, il faut savoir que **Laravel** gère l’interaction avec la base de données à travers deux concepts qui sont : **La migration et Eloquent**.

La « **Migration** » consiste à gérer les tables de la base de données de façon incrémentale c’est-à-dire que l’on peut construire la table au fur et à mesure que l’on avance dans le projet. Avec la migration, on crée une table, on la modifie, on la met à jour et ainsi de suite jusqu’avoir la structure voulu.

« **Eloquent** » est l’ORM de **Laravel** qui veut dire (**Objet Relational Mapping ou Mappage Objet Relationnel),** ce qui permet d’effectuer des opérations sur la base de donnée tout en modélisant les tables et leurs relation sous forme d’objet. Ainsi on interagit avec la base de données à travers ces modèles objets, ce qui rend ces opérations plus faciles.

Après avoir téléchargé le projet depuis GIT, renommé le projet de « **AuthRapide** » en « **TableAndOperation** » :



Commençons à présent par la première la partie qui consiste à créer les tables de la base de données.

# Création de table :

Pour créer la table **Films**, nous allons utiliser l’Outil en ligne de commande « **Artisan** », ainsi on se déplace dans le répertoire du projet et on lance la commande suivante comme sur l’image :

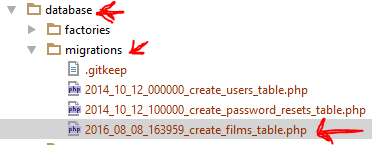


Cette commande génère un fichier « **date\_heure\_create\_films\_table.php** » dans le répertoire « **database/migrations** » qui contient une classe héritant de **Migration** où on va créer le schéma de notre table. L’option « **--create=films**  de la commande» permet de créer le schéma initiale de la table « **films** » dans notre classe migration (En absence de cette option, on se retrouve avec un fichier contenant seulement deux fonctions vides **up() & down()**). En générale, nous allons utiliser cette syntaxe pour créer une table :

**php artisan make:migration create\_<NOM TABLE>s\_table –create=<NOM TABLE>s**.

**Il est recommander d’utiliser « s » à la fin du nom de la table**, on verra pourquoi par la suite.

Ci-dessous représente le fichier contenant la migration :



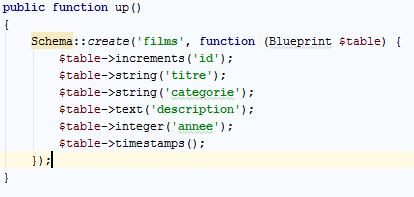
Le contenu du fichier contenant la **Migration** est la suivante:



Comme on peut le voir, le fichier contient une classe héritant de **Migration** et contient 2 méthodes à savoir « **up()** pour la création de la table » et « **down()** pour la suppression de la table ». Comme nous voulons créer une table, nous allons nous intéresse à la méthode « **up()** ». Le code déjà présent dans la méthode « **up()** » utilise ce que l’on appelle **Facade** « **Schema** permet d’effectuer des opérations sur la table [PLUS SUR L'OBJET SCHEMA](https://laravel.com/docs/master/migrations#tables)» qui appelle une méthode **create(…)** dont le premier paramètre représente le nom de la table à créer et le second est une **fonction anonyme** qui spécifie les colonnes de la table et les contraintes associer.

Modifions le contenu de la fonction « **up()** » pour y ajouter les colonnes de la table **films** **(id, titre, categorie, desctiption, annee)** en utilisant l’objet « **Blueprint** passé en paramètre de la **fonction anonyme** » ; En effet cet objet contient un ensemble de méthode qui permettent de créer une colonne avec les contraintes associer. [PLUS INFO SUR CES METHODES](https://laravel.com/docs/5.1/migrations#creating-columns).

Le contenu de la méthode doit ressembler à ci-dessous pour la création de la table :



A la suite de la modification sur le schéma de la table, on doit notifier **Laravel** de ce changement pour qu’il fasse une « **Migration** » autrement dit un genre de mise à jour de la table. Pour notifier **Laravel** de cette modification, on utilise l’outil « **Artisan** » : « **php artisan migrate** », ce qui donne :



A ce point la migration est faite et la table a été mise à jour.

NB : D’autre opération sur la base de donnée est possible telle que :

* Annulation de la dernière migration : « **php artisan migrate:rollback** ».
* Rafraichissement de toutes les migrations : « **php artisan migrate:refresh**».
* Ajouter une colonne à une table en ligne de commande :

**« php artisan add\_<NOM COLONNE>\_to\_<NOM TBALE>\_table –table=<NOM TABLE> »**

Il serait sans intérêt de créer une table si on ne peut pas interagir avec cette dernière. Passons maintenant à la prochaine étape qui consiste à interagir avec les tables de la base de données.

# Interaction avec la base de données (Opération de recherche, tri et pagination) :

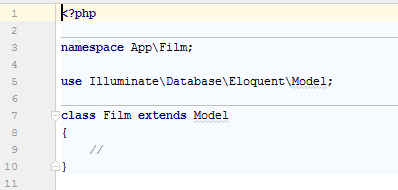
Pour effectuer les opérations de recherche, tri et la pagination sur une table, **Laravel** propose « **Eloquent** » comme introduit ci-haut, il permet d’assurer l’interaction avec la base de données en utilisant un modèle objet de la table. Donc la première étape pour l’interaction consiste à créer le modèle objet, ensuite utilisé ce modèle pour effectuer les opérations (création, recherche …).

* **Création du modèle** : Pour créer le modèle, **Laravel** propose toujours avec l’outil **artisan** une option permettant de créer le modèle. Alors dans le répertoire du projet, on lance « **php artisan make:model <Modele>** ». Par convention le nom du modèle correspond à celui de la table avec la première lettre en Majuscule et en singulier d’où le nom de la table en pluriel (Ce permet à **Laravel** d’associer le model en question à la bonne table à travers cette convention de nommage), comme recommander, on va créer le model dans un répertoire « **Film** » séparer, ce qui donne l’illustration suivante :



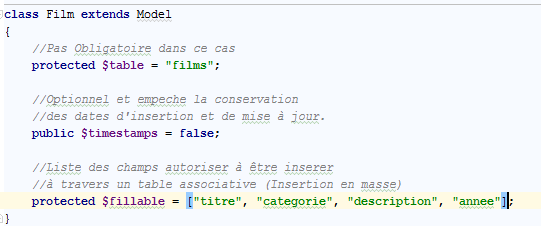
A la suite de l’exécution de cette commande, nous aurons un fichier «**app/Film/Film.php** », ce dernier représente notre modèle objet pour la table **« films »**, on l’utilisera ci-après pour effectuer des opérations sur cette table.

Avant de commencer à faire des opérations, commençons par voir le contenu du modèle en se rendant dans le fichier « **app/Film/Fim.php** » :



Alors comme nous avons suivi les conventions de nommage de la table (**nom de la table au pluriel**) et du modèle (le Modèle a même nom que la table au singulier et commence par une lettre en majuscule), **Laravel** peut déduire la table associé au modèle, dans le cas contraire, il est nécessaire d’ajouter l’attribut « **$table = ‘TableName’** » dans le modèle pour qu’il associe le modèle à la bonne table.

En plus, **Laravel** garde la date d’insertion et de mise à jour d’un élément dans une table, dans notre cas ce n’est pas vraiment nécessaire donc va donc l’enlever, pour ce faire on redéfinit une variable « **$timestamps = false ;** » de la classe Model. Finalement il faut une dernière modification dans le modèle pour effectuer l’insertion en masse (insérer un **film** avec un tableau associatif) ce qui fait que le contenue du model ressemblerais à la suivante :



NB : Il faudra éviter de mettre des champs critiques (**Numéro de compte bancaire par exemple**) dans la liste des champs assignable en masse.

* On peut maintenant commencer à interagir avec le modèle, en effet notre modèle « **Film** » hérite de la classe « **Model** » à travers laquelle on peut récupérer un objet nommé « **QueryBuilder** c’est-à-dire un constructeur de requête», cet objet contient un ensemble de méthode pour interagir avec notre table associé au modèle « **Film** ». Une liste de ces méthodes se trouve sur le lien suivant : [LISTE DES METHODES QueryBuilder](https://laravel.com/docs/master/queries#introduction).

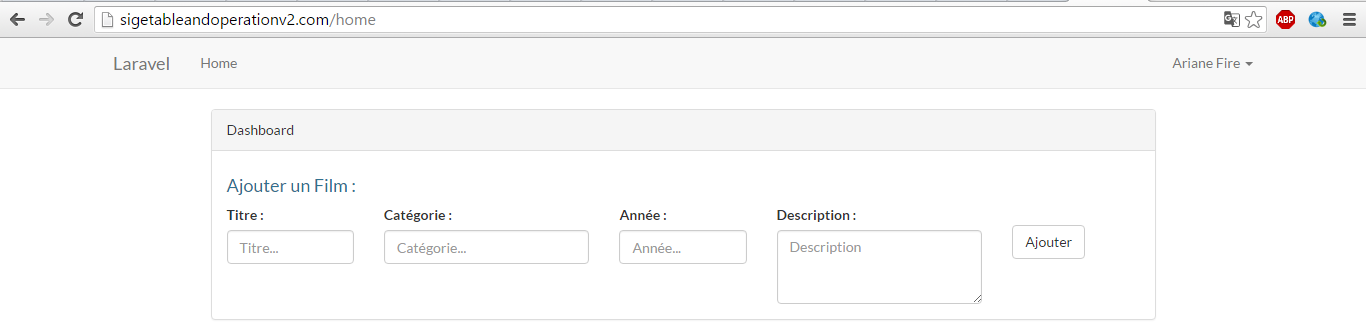
Nous allons commencer par créer un formulaire pour l’insertion d’un film.

* **Insertion d’un film:**

Pour insérer un film, il sera intéressant d’avoir un formulaire de saisie des informations sur le film. Pour ce faire nous allons éditer le fichier « **ressources/views/home.blade.php** » qui correspond à la page afficher après l’authentification d’un utilisateur. Nous allons y ajouter le formulaire suivant dans la section « **panel-body** ».

*<!--Formulaire d'insertion de Film-->*<**h4 class="text-info"**>Ajouter un Film :</**h4**>  
<**form action="/home/postFilm" method="post"**>  
 <**input type="hidden" name="\_token" value="**{{ csrf\_token() }}**"** />  
 <**div class="row"**>  
 <**div class="col-md-2"**>  
 <**label for="titre"**>Titre :</**label**>  
 <**input type="text" class="form-control" name="titre" placeholder="Titre..."** />  
 </**div**>  
  
 <**div class="col-md-3"**>  
 <**label for="categorie"**>Catégorie :</**label**>  
 <**input type="text" class="form-control" name="categorie" placeholder="Catégorie..."** />  
 </**div**>  
  
 <**div class="col-md-2"**>  
 <**label for="annee"**>Année :</**label**>  
 <**input type="number" class="form-control" name="annee" placeholder="Année..."** />  
 </**div**>  
  
 <**div class="col-md-3"**>  
 <**label for="description"**>Description :</**label**>  
 <**textarea class="form-control" name="description" rows="3" cols="40" placeholder="Description"**></**textarea**>  
 </**div**>  
  
 <**div class="col-md-1"**>  
 <**br**/>  
 <**button type="submit" class="btn btn-default"**>Ajouter</**button**>  
 </**div**>  
  
 </**div**>  
</**form**>

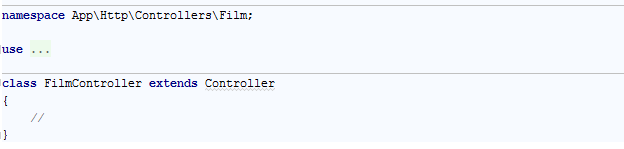
Après l’ajout du formulaire on a une vue suivante après l’authentification :



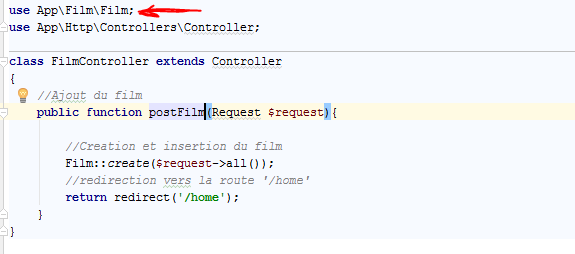
Pour gérer la soumission du formulaire, il faudra créer un **Controller** et associer la route «**/home/postVideo** » à ce Controller. Toujours étant situé dans le répertoire du projet, on lance « **php artisan make:controller Film/FilmController**» ce qui va nous créer un répertoire « **Film** » dans « **app/http/Controller** » et y placer le **Controller** «**FilmController**». Ce qui donne l’illustration suivante :



Ça fait partie des bonnes pratiques de mettre le **Controller** dans un répertoire approprié. Ci-dessous représente le contenu initial du Controller :



Nous allons ajouter une méthode qui va gérer l’ajout du film, ce qui donne :



Assurez-vous d’avoir l’instruction « **use App\Film\Film**» dans les espaces de nom utiliser en haut de la page.

Comme on peut le voir, cette fonction utilise la méthode « **create(…)**» du modèle « **Film** » pour créer et insérer le **film** **($request->all()** permet de retourner tous les paramètres du formulaire dans un tableau associatif donc les informations du **Film**) ; Ensuite la fonction redirige la requête sur la route «**/home** ».

A ce niveau si vous exécuter, l’application il va vous lance une exception car on n’a pas encore configuré de route pour gérer la soumission du formulaire donc ajoutons cette route, dans le fichier « **app/http/routes.php** » ajouter la route suivante :



Si on exécute l’application, le **film** sera bien ajouter mais il n’y’a aucun moyen de voir le **film**, aussi en insérant le **Film** on ne teste pas si les informations sur le **Film** sont valide ou pas.

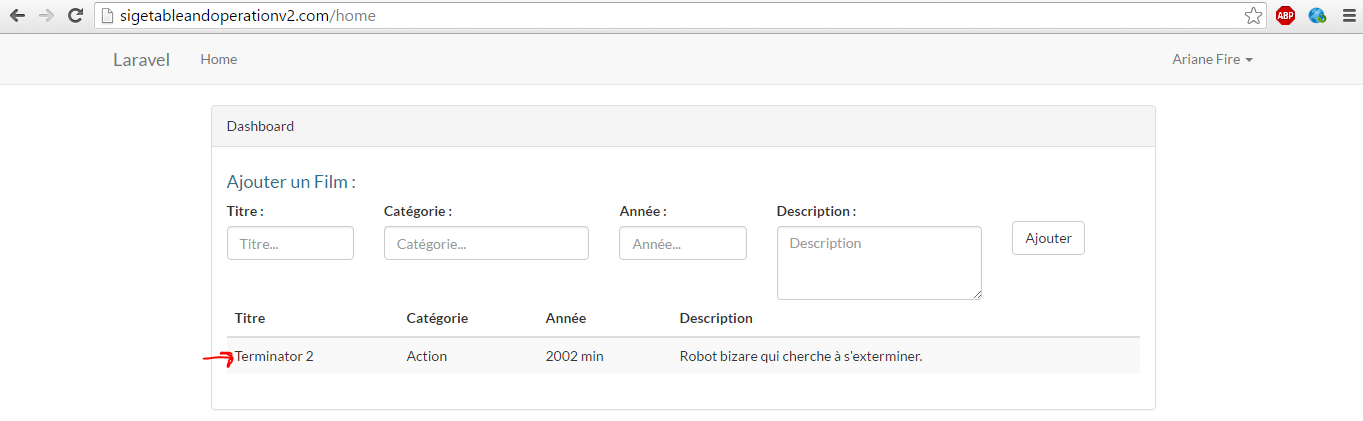
A présent réglons le problème d’affichage du **Film**, comme on redirige la requête vers la route « **/home** », on va donc modifier le **Controller** de cette route pour retourner la liste des **Films** et aussi ajouter du code dans la vue retourner par ce **Controller** pour afficher la liste des Films. Commençons par régler l’affichage de la liste de **Films** en ajoutant le code ci-dessous juste après le formulaire d’ajout de **Film** dans le fichier « **home.blade.php** » :

*<!--Affichage de la liste des videos-->*<**table class="table table-striped"**>  
 <**thead**>  
 <**tr**>  
 <**th**>Titre</**th**>  
 <**th**>Catégorie</**th**>  
 <**th**>Année</**th**>  
 <**th**>Description</**th**>  
 </**tr**>  
 </**thead**>  
 <**tbody**>  
 **@forelse(**$films **as** $film**)** <**tr**>  
 <**td**>{{$film->**titre**}}</**td**>  
 <**td**>{{$film->**categorie**}}</**td**>  
 <**td**>{{$film->**annee**}}</**td**>  
 <**td**>{{$film->**description**}}</**td**>  
 </**tr**>  
 **@empty** <**tr**>  
 <**td**>No Film.</**td**>  
 </**tr**>  
 **@endforelse** </**tbody**>  
</**table**>

Ce code parcours une variable « **$films** » et l’affiche sous forme de table donc ceci suppose que la variable « **$films** » est passé à la vue en question (**home.blade.php**), donc modifions le **Controller (HomeController.php)** de cette vue pour y ajouter le code nécessaire pour retourner la liste des films. Ce Controller devra donc ressembler à suivante au niveau de la fonction **index()** :

**public function** index()  
{  
 **return** view(**'home'**, [**'films'** => Film::*orderBy*(**'titre'**, **'asc'**)->get()]);  
}

Noter bien l’espace de nom de notre Model « **use App\Film\Film** », et aussi le table associatif doit avoir comme clé « **films** » car dans la vue on parcourt une variable nommé « **$films**». On remarque aussi à partir l’expression « **Film::orderBy(‘titre, ‘asc’)** » que la liste des films sera trié par le titre ascendant par défaut.

A ce niveau, si on exécute l’application tous devrait fonction bien à un détail près qui est la validation des informations saisie par l’utilisateur que l’on verra enfin. 

NB : Au lieu d’ajouter des films manuellement, si vous disposer d’une ressource de film, Laravel fournit un moyen de le charger dans la base de donnée.

Pour ce faire, on va créer ce que l’on appelle « **Seeder** » qui est un fichier dont le contenu pourra être exécuté par le fichier « **database/seeds/DatabaseSeeder.php** ». Créons un « **Seeder** » pour notre table Film en utilisant l’outil « **Artisan** », on exécute la commande ci-dessous pour créer le Seeder :



Ceci va générer un fichier « **database/seeds/FilmTableSeeder.php** » dans lequel on va modifier une méthode « **run()** » pour charger notre ressource de film dans la table « **films** ». On va d’abord télécharger la ressource de films (un fichier XML contenant des films) [ICI](https://github.com/ArianeExtern/Ressources.git). Après téléchargement de la ressource « **Films.xml** », on le place dans le répertoire « **app** » et on ajoute le code suivant dans la méthode « **run()** » de notre **Seeder** « **FilmTableSeeder**.php » :

**public function** run()  
{  
 *//Charger le fichier XML* $xml = simplexml\_load\_file(**"App/Films.xml"**);  
  
 *//Parcours et insertion dans la base de données* **foreach**($xml->children() **as** $film) {  
 Film::*create*([  
 **'titre'** => $film[**'titre'**],  
 **'categorie'** => $film->**GENRE**,  
 **'annee'** => $film->**ANNEE**,  
 **'description'** => $film->**PAYS** ]);  
 }  
}

Comme on peut le voir ce code charge le fichier XML, le parcours et l’insère dans la table « films ». Ensuite nous ajoutons le code ci-dessous dans le fichier « **database/seeds/DatabaseSeeders.php** » pour qu’il puisse exécuter le code ci-dessus lorsque l’on va lancer le **Seed** :

**public function** run()  
{  
 $this->call(FilmTableSeeder::***class***);  
}

Finalement on exécute la commande ci-dessous en ligne de commande pour lancer le **Seed** :



A ce niveau, si on exécute l’application, on verra la liste des Films récupérer depuis le fichier XML. A présent passons à la recherche d’un film.

* **Recherche de Film(s) :**

A un moment donné le nombre de films peut devenir énorme à point que l’on ne puisse plus retrouver un **film** donc nous allons ajouter une option de recherche de **film**.

Pour ce faire, commençons tous d’abord par ajouter un champ de recherche de film dans la page qui affiche la liste des **films**. On va donc ajouter le formulaire suivant juste au-dessus du formulaire d’ajout de **film** dans le fichier « **home.blade.php** » :

*<!--Formulaire de recherche de film-->*<**form action="/home/searchFilm" method="get"**>  
 <**div class="row"**>  
 <**div class="col-md-2"**>  
 <**h4 class="text-info"**>Rechercher : </**h4**>  
 </**div**>  
  
 <**div class="col-md-4"**>  
 <**div class="form-group-lg"**>  
 <**input type="text" class="form-control" name="keyword" placeholder="Recherche par mot clé"** />  
 </**div**>  
 </**div**>  
  
 <**div class="col-md-4"**>  
 <**button type="submit" class="btn btn-lg btn-danger"**>Rechecher</**button**>  
 </**div**>  
 </**div**>  
</**form**>

Comme on peut le voir, l’action du formulaire est redirigé vers la route « **/home/searchFilm** » donc on doit effectuer deux opérations : la première est la création de la route « **/home/searchFilm**» ensuite on passe à la création de l’action exécuter au niveau du **Controller** « **FilmController.php** ».

Pour la route, il suffit d’ajouter le code suivant dans le fichier « **routes.php** » :

Route::*get*(**'/home/searchFilm'**, [**'middleware'** => **'auth'**, **'uses'** => **'Film\FilmController@searchFilm'**]);

Créons maintenant l’action « **searchFilm()** » dans le **Controller** « **FilmController** » :

*//Recheche de la Film***public function** searchFilm(Request $request){  
 $keyword = $request->get(**'keyword'**);  
 $keyword = **'%'**.$keyword.**'%'**;  
 $films = Film::*where*(**'titre'**, **'like'**, $keyword)  
 ->orWhere(**'categorie'**, **'like'**, $keyword)  
 ->orWhere(**'annee'**, **'like'**, $keyword)  
 ->orWhere(**'description'**, **'like'**, $keyword)  
 ->orderBy(**'titre'**, **'asc'**)  
 ->get();  
  
 **return** view(**'home'**, [**'films'** => $films]);  
}

Cette action permet de vérifier si le mot clé est contenu dans l’une de colonnes de chaque ligne de la table « **films** », insensible à la casse et aussi s’il y’a pas de films trouve, il affiche le message aéquat. Tous ceci a été facilité en utilisant notre modèle « **Film** » qui contient une méthode « **where (‘Nom Colonne’, ‘like|=|>|<..’, valeur)** » qui applique le type de teste passé en 2ième argument « **‘like’ => ‘comme’, ‘=’ => ‘Egalité’ …**» sur la colonne dont le nom est passé comme 1ier argument et teste est effectué avec le 3ième argument ; Comme si c’était pas suffisant, cette méthode retourne, comme toutes les autres méthode d’un modèle, ce que l’on appelle « **QueryBuilder** » dont le but est de construit une requête **SQL** mais sous forme de fonction comme « **where(…)** » ici, ce qui fait que l’on peut enchainé plusieurs requête l’une après l’autre.

A ce niveau la recherche marche bien, vous pouvez donc ajouter autant de films et les rechercher facilement donc passons maintenant au trie.

* **Trie des films par le titre et année :**

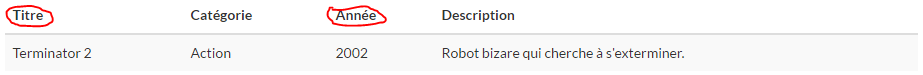
Pour mettre en place le tri, on va procéder de la même façon qu’avec la recherche c’est-à-dire :

🡪Agir sur la vue pour que nous ayons un moyen de faire le tri ;

🡪 Créer une route pour gérer le mécanisme de trie ;

🡪Et créer l’action associée à la route dans le Controller.

Commençons par agir sur la vue pour mettre l’option de trie, pour ce faire, reportons nous à l’image ci-dessous qui est l’entête de la table qui affiche les films :



Alors nous allons supposer que si l’on clique sur « **Titre**» entouré en rouge, le trie se ferais par rapport au titre et par ordre alphabétique inverse, si on clique une seconde fois le tri est dans l’ordre alphabétique et lorsque que l’on clique sur « **Année** » le même principe est appliqué.

Alors pour ce faire on va modifier le fichier « **home.blade.php** » pour rendre c’est deux élément comme des liens, ce qui fait l’entête de la table affichant les films devient :

<**thead**>  
<**tr**>  
 <**th**>  
 <**a href="**{{url(**'/home/sortFilm?titre'**)}}**"**>Titre</**a**>  
 </**th**>  
 <**th**>Catégorie</**th**>  
 <**th**>  
 <**a href="**{{url(**'/home/sortFilm?annee'**)}}**"**>Année</**a**>  
 </**th**>  
 <**th**>Description</**th**>  
</**tr**>  
</**thead**>

Alors passons à la création du **Controller** qui va gérer le tri, donc dans le fichier « **routes.php** » on aura la ligne suivante :

Route::*get*(**'/home/sortFilm'**, [**'middleware'** => **'auth'**, **'uses'** => **'Film\FilmController@sortFilm'**]);

Comme on peut le voir sur l’image cette route n’est aussi accessible que par un utilisateur connecté **(«Middleware => ‘auth ‘ »)**. Ainsi dans le Controller « **FilmController** » on aura :

*//Trie des films sophistiqués***public function** sortFilm(Request $request){  
 *//On recupere une instance de QueryBuilder* $films = Film::*query*();  
  
 **if**($request->exists(**'titre'**)){  
 $films = $films->orderBy(**'titre'**, $this->getOrder(**'titre'**));  
 }  
  
 **if**($request->exists(**'annee'**)){  
 $films = $films->orderBy(**'annee'**, $this->getOrder(**'annee'**));  
 }  
  
 **return** view(**'/home'**, [**'films'** => $films->get()]);  
}

Comme on peut le voir sur l’image, il y’a une fonction « **getOrder(…)** » qui retourne l’ordre du tri, en effet lorsqu’on trie pour la première fois, ça sera dans l’ordre « **ascendant** » et pour la deuxième fois ça sera dans l’ordre « **descendant** » et ainsi de suite, c’est ce que la fonction « **getOrder** » fait, il retourne le bon ordre en stockant l’état du tri dans la session.

*//Fonction permettant de gerer l'ordre de trie***private function** getOrder($param){  
 **if**(Session::*has*($param)){  
 Session::*put*($param, Session::*get*($param) == **'asc'** ? **'desc'**:**'asc'**);  
 }**else**{  
 Session::*put*($param, **'asc'**);  
 }  
 **return** Session::*get*($param);  
}

A ce point le trie marche donc on va s’attaquer maintenant à la partie pagination.

* **Pagination des Films :**

Pour mettre en place la pagination, le modèle « **Eloquent** » nous propose une façon très simple de le faire en utilisant la méthode « **paginate(NombreElementParPage)** » de l’objet « **QueryBuilde**r » qui est utilisé pour faire la requête sur le modèle, cette méthode retourne un objet « **LengthAwarePaginator »** que l’on peut itérer pour afficher les éléments et il possède en même temps une méthode **« render()** » que l’on utilise dans notre vu pour afficher les boutons de navigation entre les pages. Pour ce faire dans notre cas on modifie le Controller « **HomeController** » pour appeler la méthode « **paginate(…)** » et aussi au niveau de « **FilmController** ».

🡪Niveau **HomeController** : On aura la modification suivante sur le méthode « **index() »** :

**public function** index()  
{  
 **return** view(**'home'**, [**'films'** => Film::*orderBy*(**'titre'**, **'asc'**)->paginate(10)]);  
}

On a passé « 10 » comme arguments de la méthode « **paginate(10)**», ceci spécifie que l’on doit avoir 10 Films par page.

🡪Niveau « **FilmController** » : On aura la modification au niveau de deux fonctions (**postFilm() & searchFilm()) :**

*//Recheche de la Film***public function** searchFilm(Request $request){  
 $keyword = $request->get(**'keyword'**);  
 $keyword = **'%'**.$keyword.**'%'**;  
 $films = Film::*where*(**'titre'**, **'like'**, $keyword)  
 ->orWhere(**'categorie'**, **'like'**, $keyword)  
 ->orWhere(**'annee'**, **'like'**, $keyword)  
 ->orWhere(**'description'**, **'like'**, $keyword)  
 ->orderBy(**'titre'**, **'asc'**)  
 ->paginate(10); *//Pagination 10 films par page* **return** view(**'home'**, [**'films'** => $films]);  
}  
  
*//Trie des films sophistiqués***public function** sortFilm(Request $request){  
 *//On recupere une instance de QueryBuilder* $films = Film::*query*();  
  
 **if**($request->exists(**'titre'**)){  
 $films = $films->orderBy(**'titre'**, $this->getOrder(**'titre'**));  
 }  
  
 **if**($request->exists(**'annee'**)){  
 $films = $films->orderBy(**'annee'**, $this->getOrder(**'annee'**));  
 }  
  
 *//Pagination de 10 Films par page* **return** view(**'/home'**, [**'films'** => $films->paginate(10)]);  
}

🡪Au niveau de la page « **Home.blade.php**» : on ajouter la ligne ci-dessous en bas de la table qui affiche la liste des films :

{!! $films->render() !!}

A ce niveau la pagination doit aussi marcher bien, finalement on va juste ajouter la validation du film avant son insertion.

# Validation des données à insertion dans une table (Intégrité des données):

Comme dans tous systèmes robustes, il sera intéressant de mettre en place un moyen de vérification des données passée à la base de données. Et pour ce faire, **Laravel** propose une classe permettant de faire la validation en précisant les règles que chaque champ du formulaire utiliser pour faire l’insertion.

On va donc ajouter une méthode « **validateFilm(…)** » dans le Controller « **FilmController** » ce qui donne :

*//Validation des données***public function** validateFilm($array){  
 **return** Validator::*make*($array, [  
 **'titre'** => **'required|min:3'**,  
 **'categorie'** => **'required'**,  
 **'annee'** => **'required'**,  
 **'description'** => **'required'**,  
 ]);  
}

Dans ce cas les règles de validation sont assez simples, en effet on a des champs obligatoires « **require** », et le titre doit avoir au minimum 3 caractères. Pour une documentions plus élaborer des règles de validation [VOIR ICI](https://laravel.com/docs/5.1/validation#available-validation-rules).

Ensuite nous allons appeler ce validateur dans la méthode qui ajoute la vidéo « **postFilm()** » du Controller « **FilmController** », donc cette méthode sera comme suit :

*//Ajout du film***public function** postFilm(Request $request){  
  
 *//Validation des données* $validator = $this->validateFilm($request->all());  
 **if**($validator->fails()){  
 **return** redirect(**'/home'**);  
 }

………

}

A ce point tous devra bien marcher.

**Conclusion :**

Pour récupérer ce projet l’exécuter, il faut :

🡪Télécharger depuis **GitHub** sur le lien : [TableAndOperationV2](https://github.com/ArianeExtern/EtudeLaravel/tree/Table).

🡪Si vous êtes sur « **Vagrant** »

🡪Déplacez-vous dans le répertoire du projet.

🡪Exécuter la commande : **vagrant up**.

🡪Vérifier le fichier « **Homestead.yaml** » et configurer le fichier « C:/Windows/etc/drivers/hosts » pour associer l’hôte à l’adresse IP.