

# LRAVEL 9

- ANPT
- Formateur: Bahmed Kamel
- [kamel.bahmed.info@gmail.com](mailto:kamel.bahmed.info@gmail.com)

# PLAN

- Revision php POO
- Laravel :
  1. Designe pattern
  2. MVC
  3. Framwork
  4. Présentation générale
  5. 2. Installation et organisation
  6. 3. Routes
  7. 4. Middlewares
  8. 5. Controller
  9. 6. Les vues blades
  10. 7. Migration et Eloquents
  11. 8. Formulaires
  12. 9. La validation
  13. 10. Authentification
  14. Projet

# PHP POO REVISION

- Poo definition
- Classes et objets
- Heritage
- Methode statique
- namespace

# POO ?

- POO signifie Programmation Orientée Objet.
- La programmation procédurale consiste à écrire des procédures ou des fonctions qui effectuent des opérations sur les données, tandis que la programmation orientée objet consiste à créer des objets qui contiennent à la fois des données et des fonctions.
- La programmation orientée objet présente plusieurs avantages par rapport à la programmation procédurale :
  1. La POO est plus rapide et plus facile à exécuter
  2. La POO fournit une structure claire pour les programmes
  3. La POO aide à garder le code PHP SEC "Ne vous répétez pas", et facilite la maintenance, la modification et le débogage du code
  4. La POO permet de créer des applications entièrement réutilisables avec moins de code et un temps de développement plus court

# CLASSES ET OBJETS

- Une classe est un modèle pour les objets et un objet est une instance de classe

```
class Fruit {  
    // Properties  
    public $name;  
    public $color;  
  
    // Methods  
    function set_name($name)  
    {  
        $this->name = $name;  
    }  
    function get_name() {  
        return $this->name;  
    }  
}
```

```
$apple = new Fruit();  
$banana = new Fruit();  
$apple->set_name('Apple');  
$banana->set_name('Banana');  
  
echo $apple->get_name();  
echo "<br>";  
echo $banana->get_name();
```

# HÉRITAGE

- Héritage en POO = Lorsqu'une classe dérive d'une autre classe
- La classe enfant héritera de toutes les propriétés et méthodes publiques et protégées de la classe parent. De plus, il peut avoir ses propres propriétés et méthodes

```
class Fruit {  
    public $name;  
    public $color;  
    public function __construct($name, $color)  
    {  
        $this->name = $name;  
        $this->color = $color;  
    }  
    public function intro() {  
        echo "The fruit is {$this->name} and the  
        color is {$this->color}.";  
    }  
}
```

```
class Strawberry extends Fruit {  
    public function message() {  
        echo "Am I a fruit or a berry? ";  
    }  
}  
$strawberry = new Strawberry("Strawberry", "red");  
$strawberry->message();  
$strawberry->intro();
```

# MÉTHODES STATIQUES

- Les méthodes statiques peuvent être appelées directement - sans créer d'abord une instance de la classe.

```
class greeting {  
    public static function welcome() {  
        echo "Hello World!";  
    }  
}
```

```
// Call static method  
greeting::welcome();
```

# NAMESPACE

- Les espaces de noms sont des qualificatifs qui résolvent deux problèmes différents :
  1. Ils permettent une meilleure organisation en regroupant des classes qui travaillent ensemble pour effectuer une tâche
  2. Ils permettent d'utiliser le même nom pour plus d'une classe
- Pour prendre un exemple concret, vous pouvez considérer que les espaces de noms fonctionnent comme les dossiers sur notre ordinateur. Alors qu'il est impossible de stocker deux fichiers de même nom dans un même dossier, on va tout à fait pouvoir stocker deux fichiers de même nom dans deux dossiers différents. En effet, dans ce dernier cas, il n'y a plus d'ambiguïté puisque les deux fichiers ont un chemin d'accès et de fait une adresse différente sur notre ordinateur



```
namespace Tutoriel\HTML;  
class Form { // ... }
```

```
use \Tutoriel\Html\Form ;  
new Form();
```

- namespace, permet de dire "je travaille dans ce dossier"
- use, permet d'importer une class d'un autre "dossier"

# MVC

- Les **design patterns** sont des solutions typiques à des problèmes communs en développement logiciel : ils ne sont pas une implémentation concrète d'une solution à un problème, mais plutôt une stratégie à appliquer pour le résoudre de façon élégante et maintenable
- Un des plus célèbres *design patterns* s'appelle MVC, qui signifie **Modèle - Vue – Contrôleur**
- Le pattern MVC permet de bien organiser son code source. Il va vous aider à savoir quels fichiers créer, mais surtout à définir leur rôle. Le but de MVC est justement de séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts

# Modèle

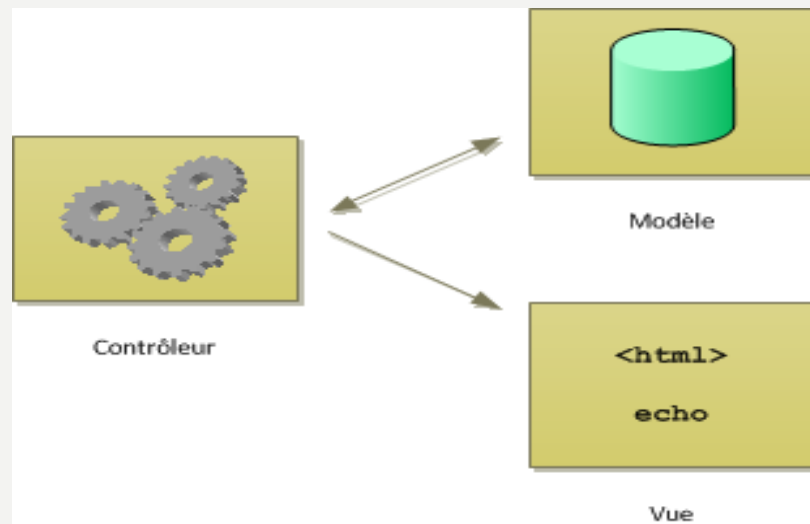
- cette partie gère les *données* de votre site. Son rôle est d'aller récupérer les informations « brutes » dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur. On y trouve donc entre autres les requêtes SQL.

# Vue

- cette partie se concentre sur l'*affichage*. Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher. On y trouve essentiellement du code HTML mais aussi quelques boucles et conditions PHP très simples, pour afficher par exemple une liste de messages

# Contrôleur

- cette partie gère la logique du code qui prend des *décisions*. C'est en quelque sorte l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue. Le contrôleur contient exclusivement du PHP. C'est notamment lui qui détermine si le visiteur a le droit de voir la page ou non (gestion des droits d'accès)



# LARAVEL

# FRAMEWORK

- un framework informatique est un « ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel ». Autrement dit une base homogène avec des briques toutes prêtes à disposition. Il existe des frameworks pour tous les langages de programmation et en particulier pour PHP
- L'utilité d'un framework est d'éviter de passer du temps à développer ce qui a déjà été fait par d'autres souvent plus compétents et qui a en plus été utilisé et validé par de nombreux utilisateurs. On peut imaginer un framework comme un ensemble d'outils à disposition. Par exemple je dois faire du routage pour mon site, je prends un composant déjà tout prêt et qui a fait ses preuves et je l'utilise : gain de temps, fiabilité, mise à jour si nécessaire

# ENVIRONNEMENT DE DÉVELOPPEMENT

- Pour fonctionner Laravel a besoin d'un certain environnement :
- un serveur,
- PHP,
- MySql,
- Composer

# INSTALLER COMPOSER

- 1 telecharger depuis <https://getcomposer.org/>
- 2 dans le cmd executer :composer global require laravel/installer

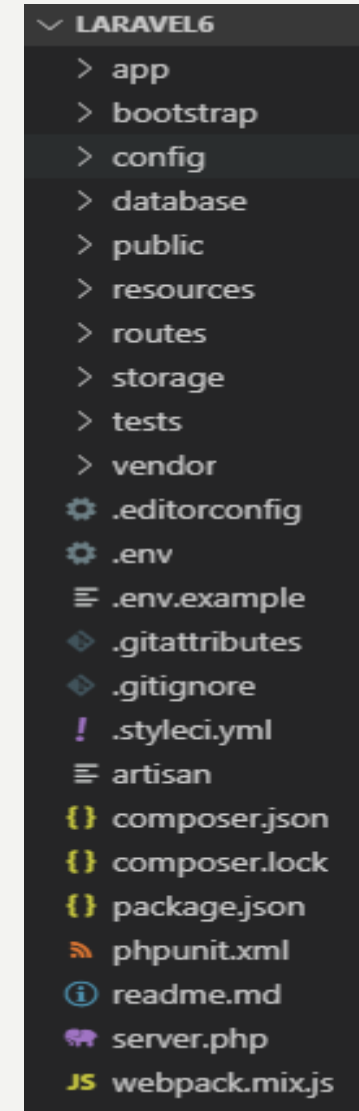
# CRÉER UN PROJET LARAVEL

- `composer create-project laravel/laravel:^9.0 example-app`



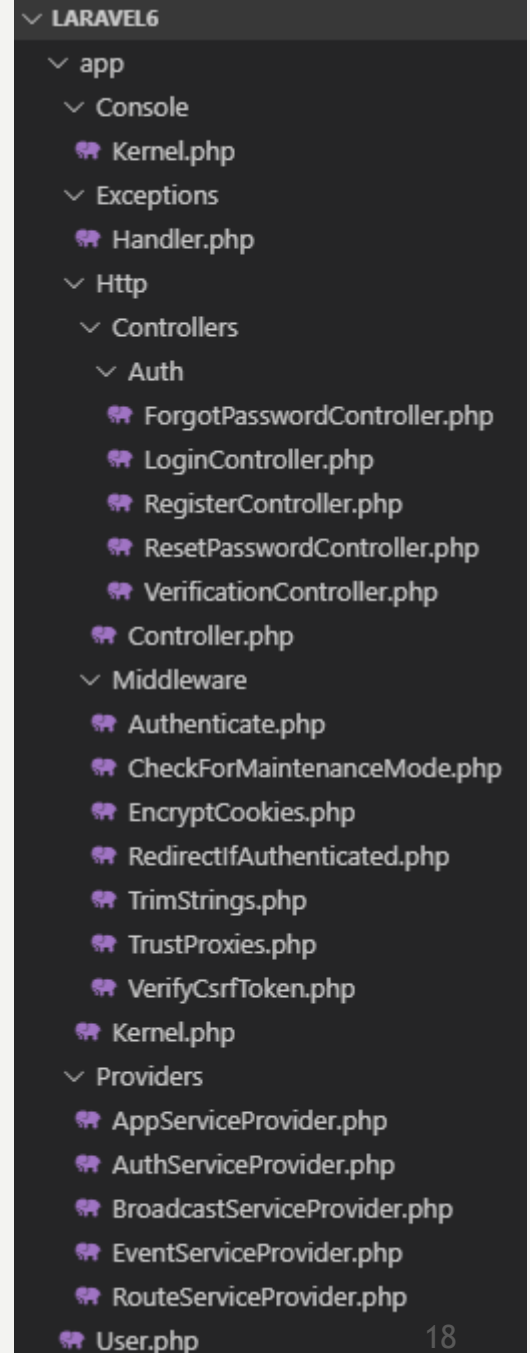
# ORGANISATION DE LARAVEL

- Lancer le serveur : `php artisan serve`



# DOSSIER APP

- Ce dossier contient les éléments essentiels de l'application :
- **Console** : toutes les commandes en mode console,
- **Http** : tout ce qui concerne la communication : contrôleurs, middlewares (il y a 7 middlewares de base qui servent à filtrer les requêtes HTTP) et le kernel,
- **Providers** : tous les fournisseurs de services (providers), il y en a déjà 5 au départ. Les providers servent à initialiser les composants.
- **User** : un modèle qui concerne les utilisateurs pour la base de données



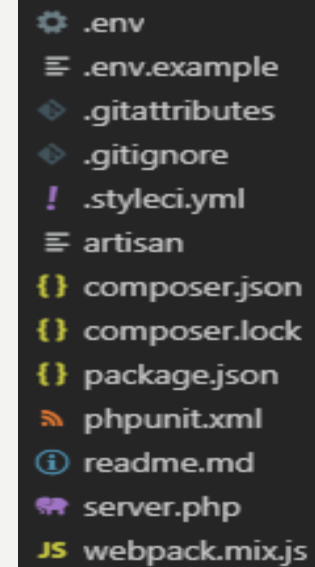
# AUTRES DOSSIERS

- **bootstrap** : scripts d'initialisation de Laravel pour le chargement automatique des classes, la fixation de l'environnement et des chemins, et pour le démarrage de l'application,
- **public** : tout ce qui doit apparaître dans le dossier public du site : images, CSS, scripts...
- **config** : toutes les configurations : application, authentification, cache, base de données, espaces de noms, emails, systèmes de fichier, session...
- **database** : migrations et populations,
- **resources** : vues, fichiers de langage et assets (par exemple les fichiers Sass),
- **routes** : la gestion des urls d'entrée de l'application,
- **storage** : données temporaires de l'application : vues compilées, caches, clés de session...
- **tests** : fichiers de tests unitaires,
- **vendor** : tous les composants de Laravel et de ses dépendances (créé par composer).

```
> bootstrap
> config
> database
> public
> resources
> routes
> storage
> tests
> vendor
```

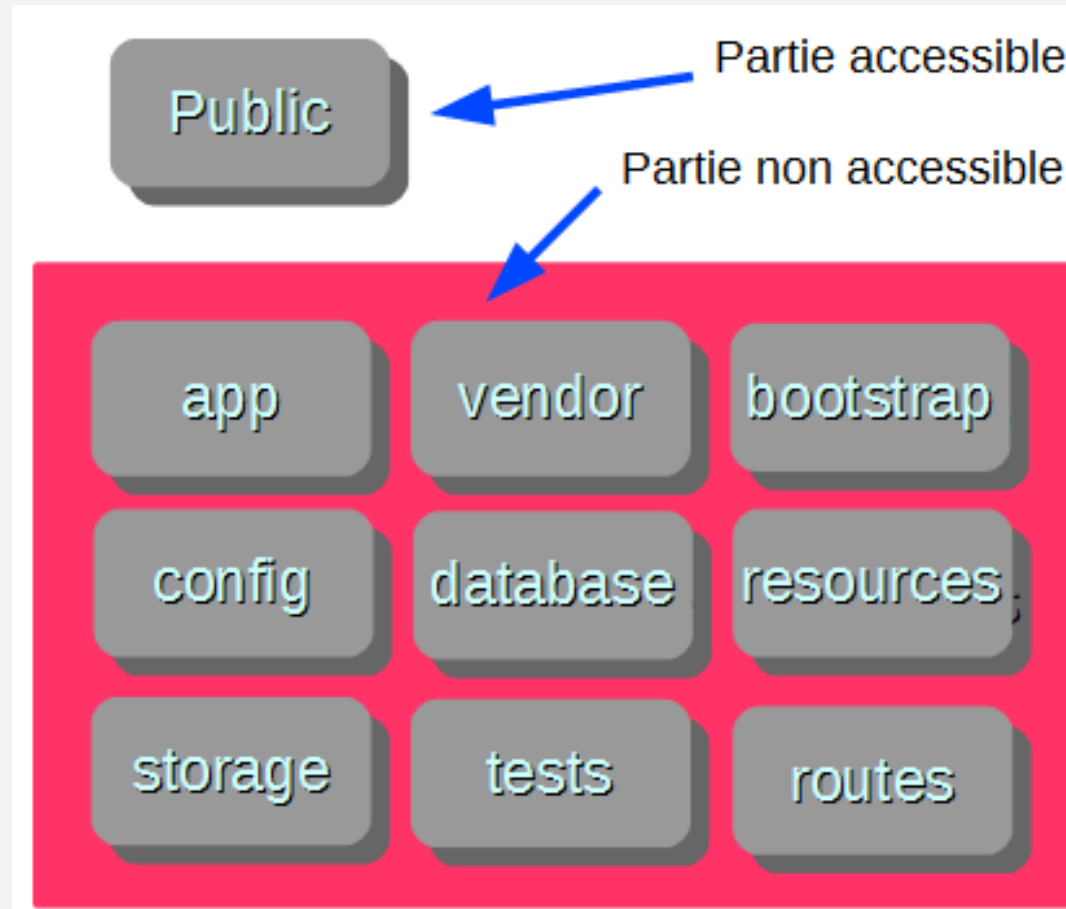
# FICHIERS DE LA RACINE

- **artisan** : outil en ligne de Laravel pour des tâches de gestion,
- **composer.json** : fichier de référence de composer,
- **package.json** : fichier de référence de npm pour les assets,
- **phpunit.xml** : fichier de configuration de phpunit (pour les tests unitaires),
- **.env** : fichier pour spécifier l'environnement d'exécution.



```
• .env  
• .env.example  
• .gitattributes  
• .gitignore  
• .styleci.yml  
• artisan  
• composer.json  
• composer.lock  
• package.json  
• phpunit.xml  
• readme.md  
• server.php  
• webpack.mix.js
```

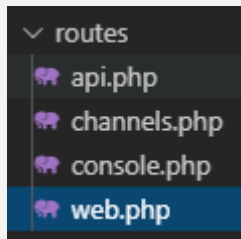
# ACCESSIBILITÉ



# LE ROUTAGE

- Les méthodes
- Il est indispensable de connaître les principales méthodes du HTTP :
- **GET** : c'est la plus courante, on demande une ressource qui ne change jamais, on peut mémoriser la requête, on est sûr d'obtenir toujours la même ressource,
- **POST** : elle est aussi très courante, là la requête modifie ou ajoute une ressource, le cas le plus classique est la soumission d'un formulaire (souvent utilisé à tort à la place de PUT),
- **PUT** : on ajoute ou remplace complètement une ressource,
- **PATCH** : on modifie partiellement une ressource (donc à ne pas confondre avec PUT),
- **DELETE** : on supprime une ressource.

- Lorsque la requête atteint le fichier **public/index.php** l'application Laravel est créée et configurée et l'environnement est détecté. Nous reviendrons plus tard plus en détail sur ces étapes. Ensuite le fichier **routes/web.php** est chargé. Voici l'emplacement de ce fichier :
- C'est avec ce fichier que la requête va être analysée et dirigée. Regardons ce qu'on y trouve au départ :

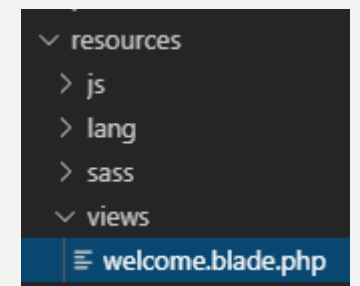


```
Route::get('/', function () {  
    return view('welcome');  
});
```

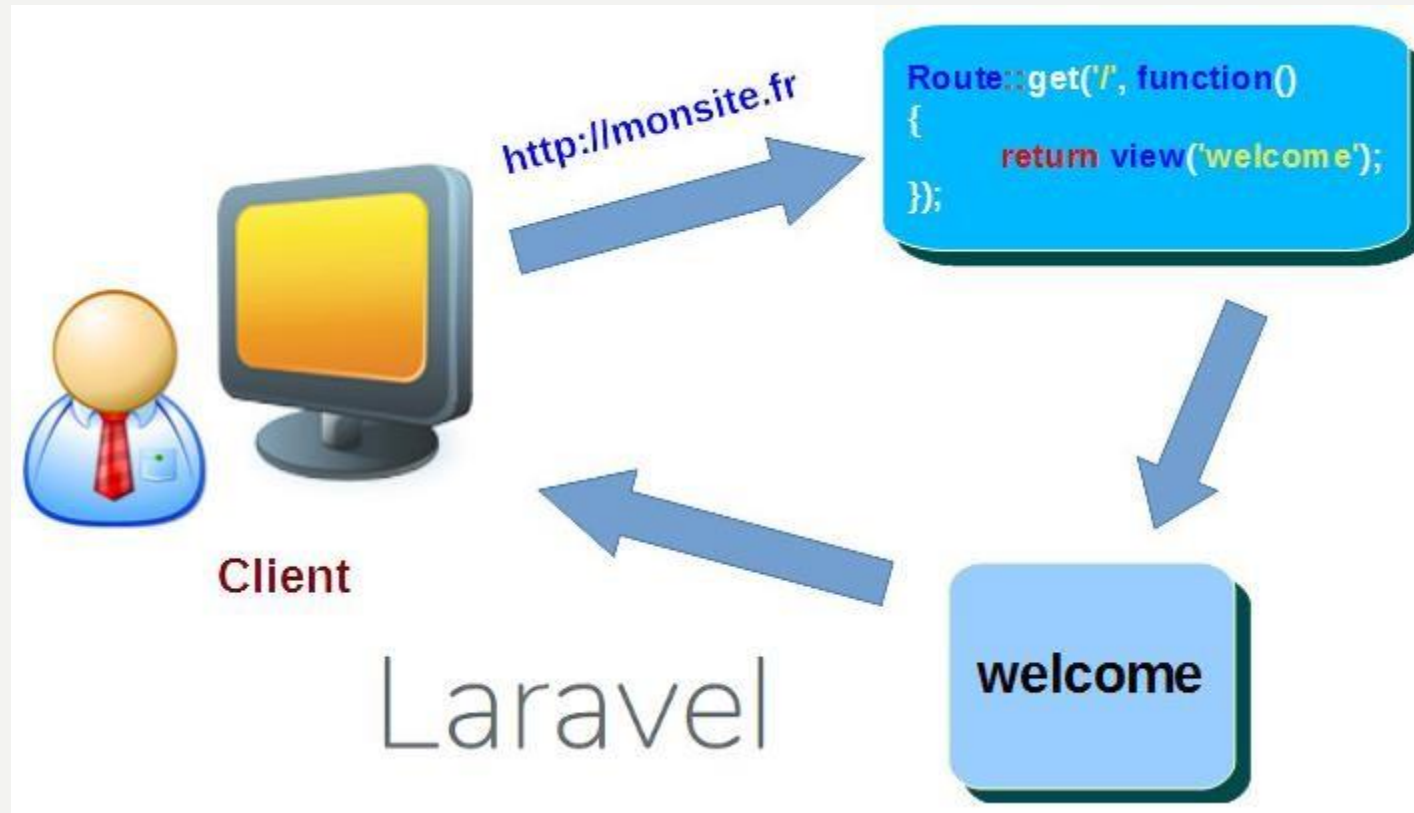
Comme Laravel est explicite vous pouvez déjà deviner à quoi sert ce code :

- **Route** : on utilise le routeur,
- **get** : on regarde si la requête a la méthode « get »,
- **/'** : on regarde si l'url comporte uniquement le nom de domaine,
- dans la fonction anonyme on retourne (**return**) une vue (**view**) à partir du fichier « welcome ».

Ce fichier « welcome » se trouve bien rangé dans le dossier des vues :



- C'est ce fichier comportant du code Html qui génère le texte d'accueil que vous obtenez au démarrage initial de Laravel





# PLUSIEURS ROUTES ET PARAMÈTRE DE ROUTE

- A l'installation Laravel a une seule route qui correspond à l'url de base composée uniquement du nom de domaine. Voyons maintenant comment créer d'autres routes. Imaginons que nous ayons 3 pages qui doivent être affichées avec ces urls :

1. <http://monsite.fr/1>
2. <http://monsite.fr/2>
3. <http://monsite.fr/3>

```
Route::get('/1', function() { return 'Je suis la page 1 !'; });  
Route::get('/2', function() { return 'Je suis la page 2 !'; });  
Route::get('/3', function() { return 'Je suis la page 3 !'; });
```

- Cette fois je n'ai pas créé de vue parce que ce qui nous intéresse est uniquement une mise en évidence du routage, je retourne donc directement la réponse au client. Visualisons cela pour la page 1 :



```
Route::get('{n}', function($n) {  
    return 'Je suis la page ' . $n . ' !';  
});
```

# ORDRE DES ROUTE

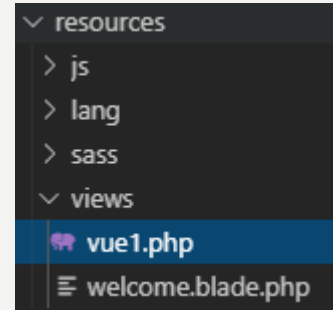


```
1 Route::get('{n}', function($n) {  
2     return 'Je suis la page ' . $n . ' !';  
3 });  
4 Route::get('contact', function() {  
5     return "C'est moi le contact.";  
6 });  
7  
8  
9
```

# LES VUE

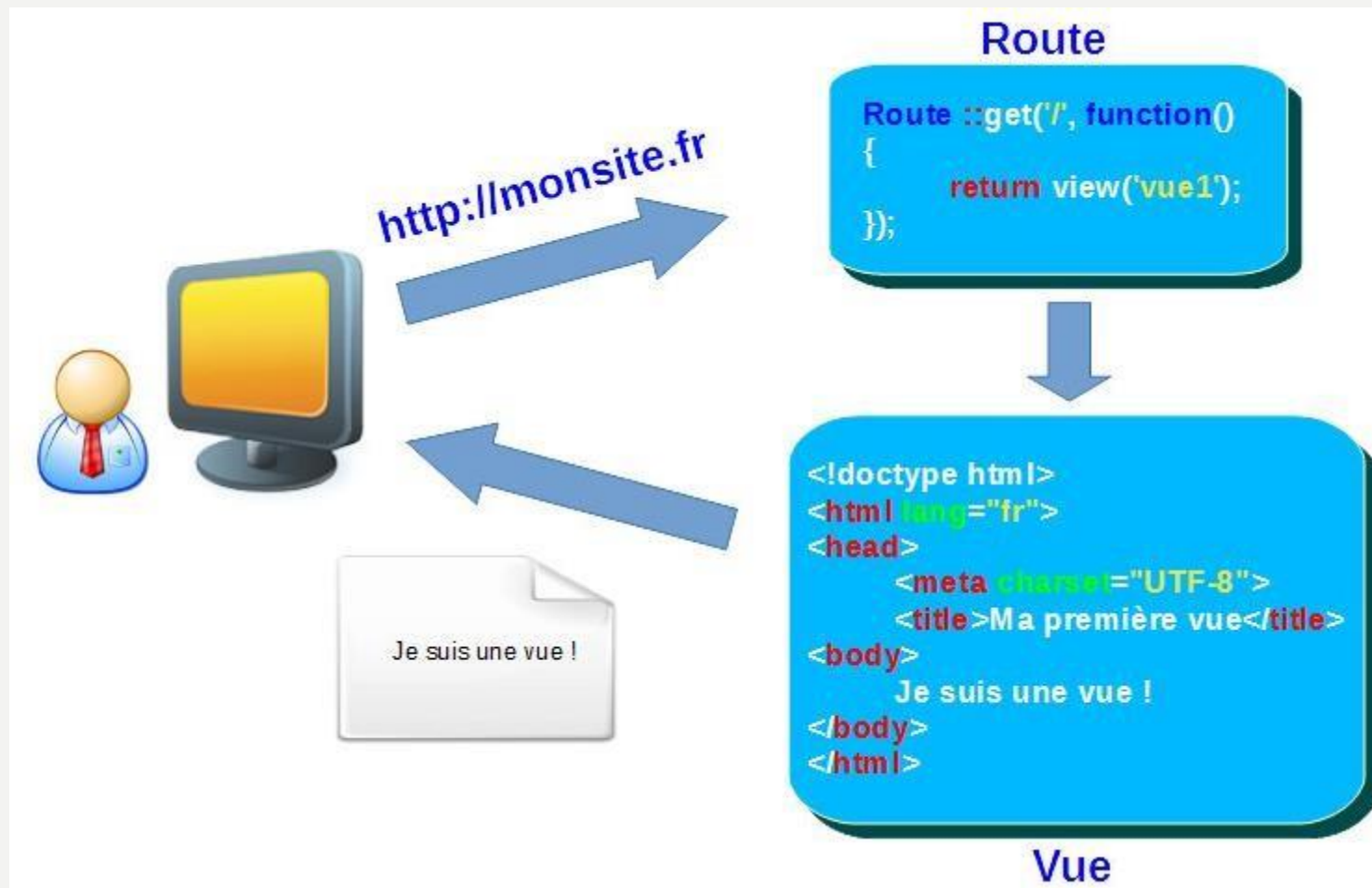
- Dans une application réelle vous retournerez rarement la réponse directement à partir d'une route, vous passerez au moins par une vue. Dans sa version la plus simple une vue est un simple fichier avec du code Html

```
<!doctype html>
<html lang="fr">
<head>
<meta charset="UTF-8">
<title>Ma première vue</title>
</head>
<body>
Je suis une vue !
</body>
</html>
```



On peut appeler cette vue à partir d'une route avec ce code

```
Route::get('/', function() {
    return view('vue1');
});
```



# VUE PARAMÉTRÉE

- En général on a des informations à transmettre à une vue, voyons à présent comment mettre cela en place. Supposons que nous voulions répondre à ce type de requête :

`http://monsite.fr/article/n`

Base de l'url

Partie fixe

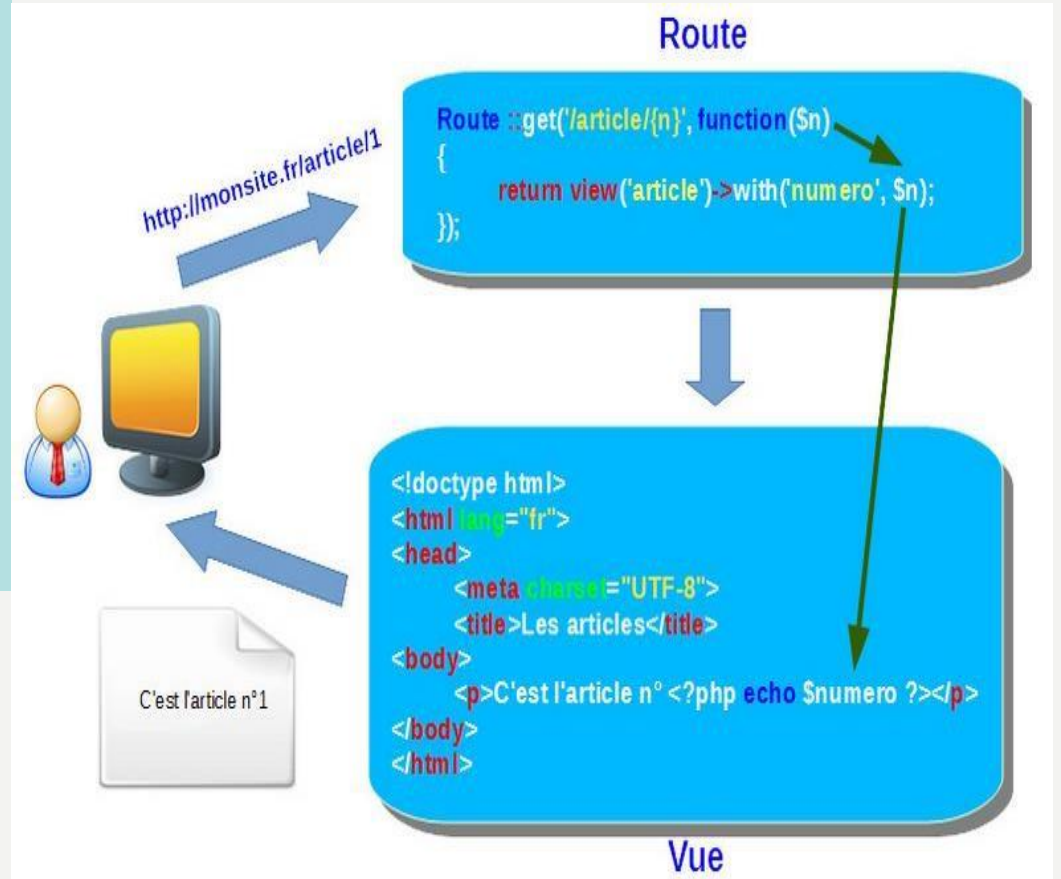
Partie variable

Il nous faut une route pour intercepter ces urls :

```
Route::get('article/{n}', function($n) {  
    return view('article')->with('numero', $n);  
})
```

Il ne nous reste plus qu'à créer la vue **article.php** dans le dossier **resources/views** :

```
<!doctype html>
<html lang="fr">
<head>
<meta charset="UTF-8">
<title>Les articles</title>
</head>
<body>
<p>C'est l'article n° <?php echo $numero ?></p>
</body>
</html>
```



# BLADE

- Laravel possède un moteur de template élégant nommé Blade qui nous permet de faire pas mal de choses. La première est de nous simplifier la syntaxe. Par exemple au lieu de la ligne suivante que nous avons prévue dans la vue précédente :

```
<p>C'est l'article n° <?php echo $numero ?></p>
```

```
<p>C'est l'article n° {{ $numero }}</p>
```

- Il suffit d'ajouter « blade » avant l'extension « php ». Vous pouvez tester l'exemple précédent avec ces modifications et vous verrez que tout fonctionne parfaitement avec une syntaxe épurée.



# UN TEMPLATE

- Une fonction fondamentale de Blade est de permettre de faire du templating, c'est à dire de factoriser du code de présentation. Poursuivons notre exemple en complétant notre application avec une autre route chargée d'intercepter des urls pour des factures. Voici la route :

```
Route::get('facture/{n}', function($n) {  
    return view('facture')->withNumero($n);  
});
```

*On se rend compte que cette vue est pratiquement la même que celle pour les articles. Il serait intéressant de placer le code commun dans un fichier.*

```
<!doctype html>  
<html lang="fr">  
<head>  
<meta charset="UTF-8">  
<title>Les factures</title>  
</head>  
<body>  
<p>C'est la facture n° {{ $numero }}</p>  
</body>  
</html>
```

- Voici le template :

```
<!doctype html>
<html lang="fr">
<head>
<meta charset="UTF-8">
<title>@yield('titre')</title>
</head>
<body>
@yield('contenu')
</body>
</html>
```

J'ai repris le code commun et prévu deux emplacements repérés par le mot clé **@yield** et nommés « titre » et « contenu ». Il suffit maintenant de modifier les deux vues. :

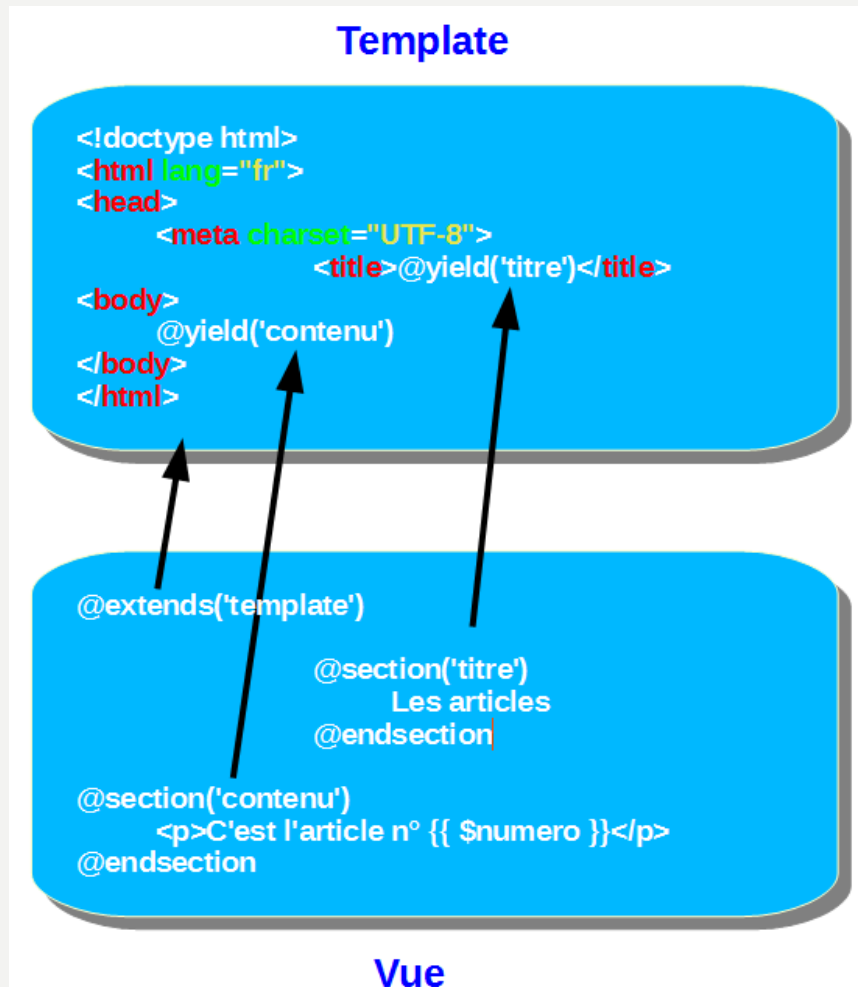
```
@extends('template')
@section('titre')
Les articles
@endsection
@section('contenu')
<p>C'est l'article n° {{ $numero }}</p>
@endsection
```

Voilà pour les articles :

Et voilà pour les factures :

```
@extends('template')
@section('titre')
Les factures
@endsection
@section('contenu')
<p>C'est la facture n° {{ $numero }}</p>
@endsection
```

Dans un premier temps on dit qu'on veut utiliser le template avec **@extends** et le nom du template « template ». Ensuite on remplit les zones prévues dans le template grâce à la syntaxe **@section** en précisant le nom de l'emplacement et en fermant avec **@endsection**. Voici un schéma pour bien visualiser tout ça avec les articles :

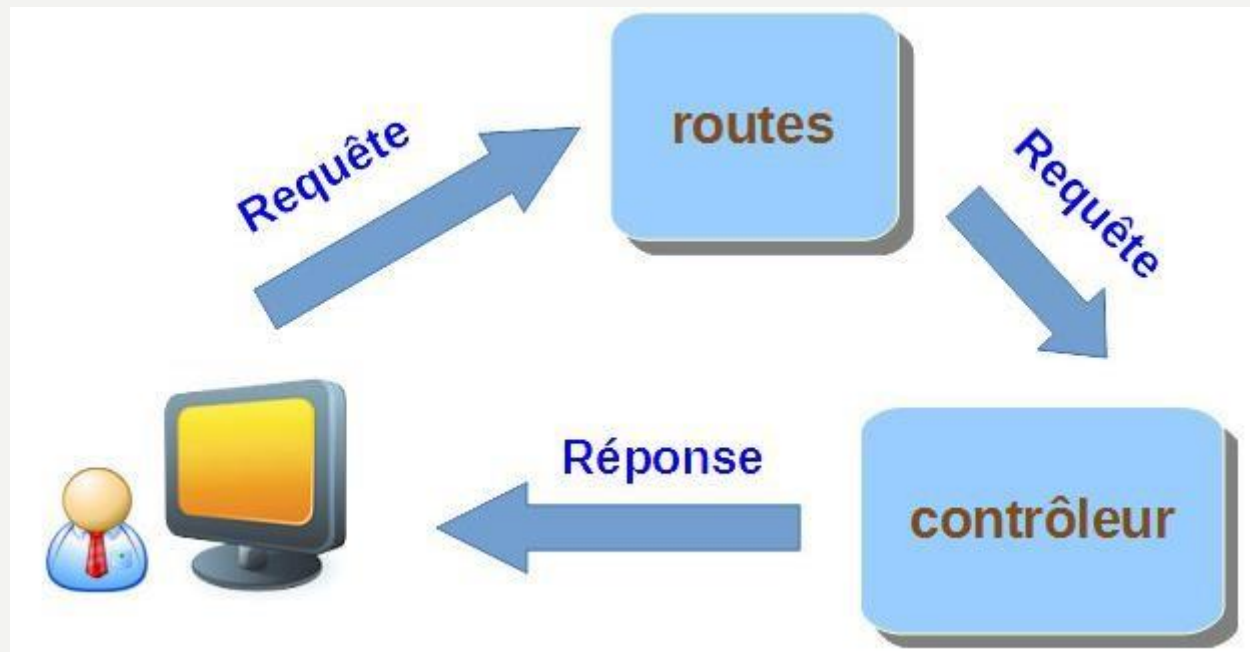


Au niveau du dossier des vues on a donc les trois fichiers :

```
views
├── article.blade.php
├── facture.blade.php
└── template.blade.php
```

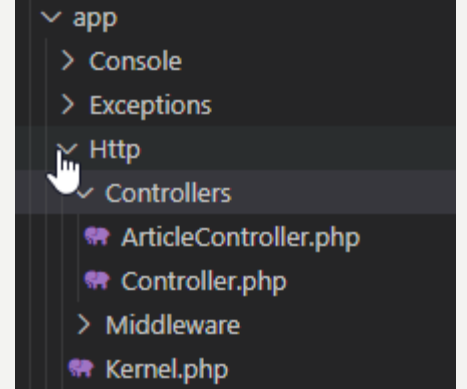
# LES CONTRÔLEURS

- La tâche d'un contrôleur est de réceptionner une requête (qui a déjà été sélectionnée par une route) et de définir la réponse appropriée, rien de moins et rien de plus. Voici une illustration du processus :



Pour créer un contrôleur nous allons utiliser Artisan.  
Dans la console entrez cette commande :

```
php artisan make:controller ArticleController
```



Ajoutez la méthode **show** :

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6
7 class ArticleController extends Controller
8 {
9     //
10 }
11
```

```
<?php
...
class ArticleController extends Controller
{
    public function show($n)
    {
        return view('article')->with('numero', $n);
    }
}
```

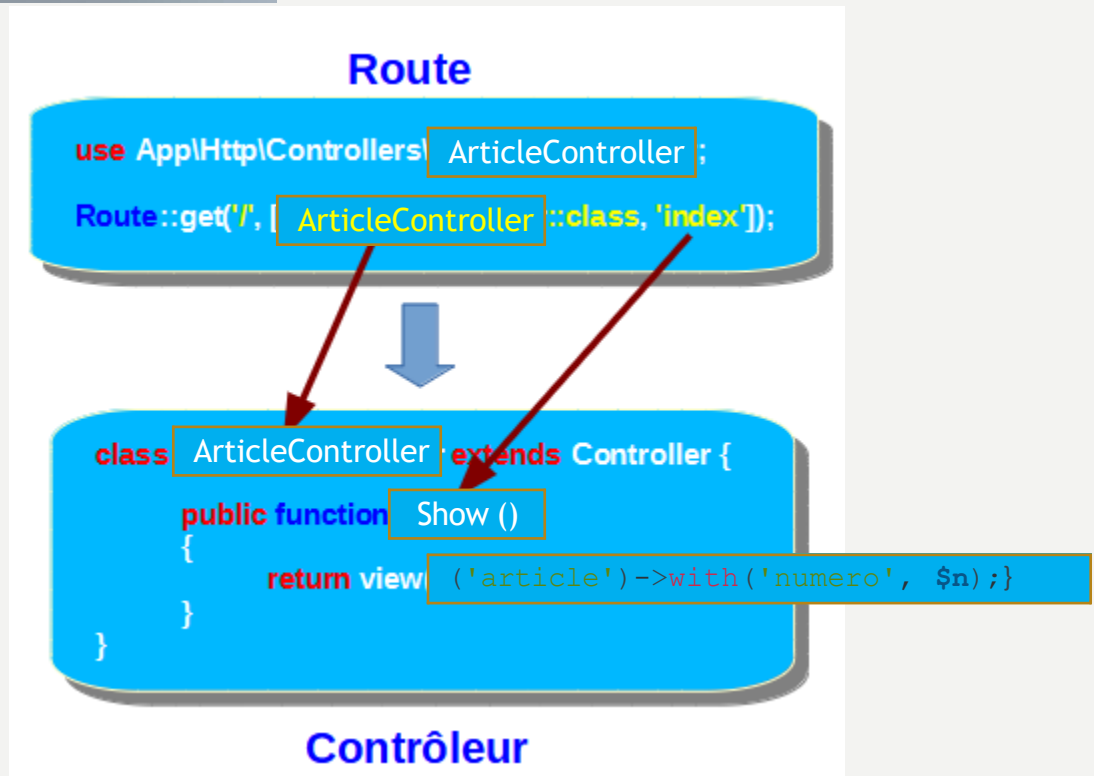
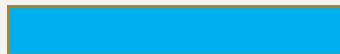
- on trouve en premier l'espace de nom (**App\Http\Controllers**),
- le contrôleur hérite de la classe **Controller** qui se trouve dans le même dossier et qui permet de factoriser des actions communes à tous les contrôleurs,
- on trouve enfin la méthode **show** qui renvoie quelque chose que maintenant vous connaissez : une vue, en l'occurrence « Article » dont nous avons déjà parlé.



```
1 use App\Http\Controllers\ArticleController;  
2 use Illuminate\Support\Facades\Route;  
3  
4  
5  
6 Route::get('/article/{n}', [ArticleController::class, 'show']);
```

*Maintenant la question qu'on peut se poser est : comment s'effectue la liaison entre les routes et les contrôleurs ?*  
Ouvrez le fichier des routes et entrez ce code :

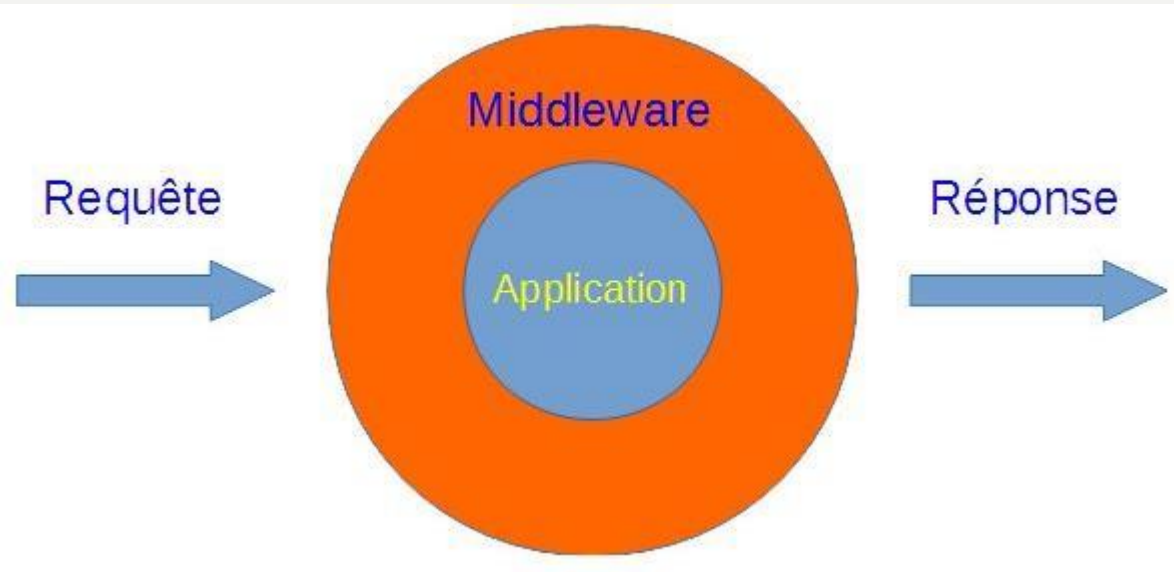
Voici une visualisation de la liaison entre la route et le contrôleur :



# MIDDLEWARE

Pour faire simple, ce sont des fonctions qui permettent d'enrichir ou de vérifier une requête avant qu'elle arrive à votre controller.

Plusieurs middlewares sont fournis de base avec Laravel comme `auth` pour autoriser l'accès à la route seulement aux requêtes liées à un utilisateur connecté.



- Les middlewares en Laravel sont des fonctions intermédiaires qui sont exécutées avant ou après une requête HTTP. Ils permettent de filtrer les requêtes, de vérifier l'authentification de l'utilisateur, de valider les entrées utilisateur, de modifier les en-têtes HTTP, etc.
- Prenons un exemple simple : lorsque vous vous connectez à un site web, le middleware vérifie votre nom d'utilisateur et votre mot de passe avant de vous permettre d'accéder au contenu protégé. Si vous n'êtes pas authentifié, vous êtes redirigé vers la page de connexion.
- En bref, les middlewares sont des outils puissants pour gérer le flux de trafic entre votre application et ses utilisateurs. Ils vous permettent de contrôler les requêtes et les réponses de manière centralisée, et de rendre votre application plus sûre et plus performante.



# FORMULAIRE

- Nous allons envisager un petit scénario avec une demande de formulaire de la part du client, sa soumission et son traitement :



On va donc avoir besoin de deux routes :

- une pour la demande du formulaire avec une méthode **get**,
- une pour la soumission du formulaire avec une méthode **post**.

- On va donc créer ces deux routes dans le fichier **routes/web.php** :

```
Route::get('/users', [UserController::class, 'create']);  
Route::post('/users', [UserController::class, 'store']);
```

- Pour faire les choses correctement nous allons prévoir un template **resources/views/template.blade.php** :

```
<!doctype html>
<html lang="fr">
<head>
<meta charset="UTF-8">
</head>
<body>
@yield('contenu')
</body>
</html>
```

Et une vue **resources/views/infos.blade.php** qui utilise ce template :

```
@extends('template')
@section('contenu')
<form action="{{ url('users') }}" method="POST">
@csrf
<label for="nom">Entrez votre nom : </label>
<input type="text" name="nom" id="nom">
<input type="submit" value="Envoyer !">
</form>
@endsection
```

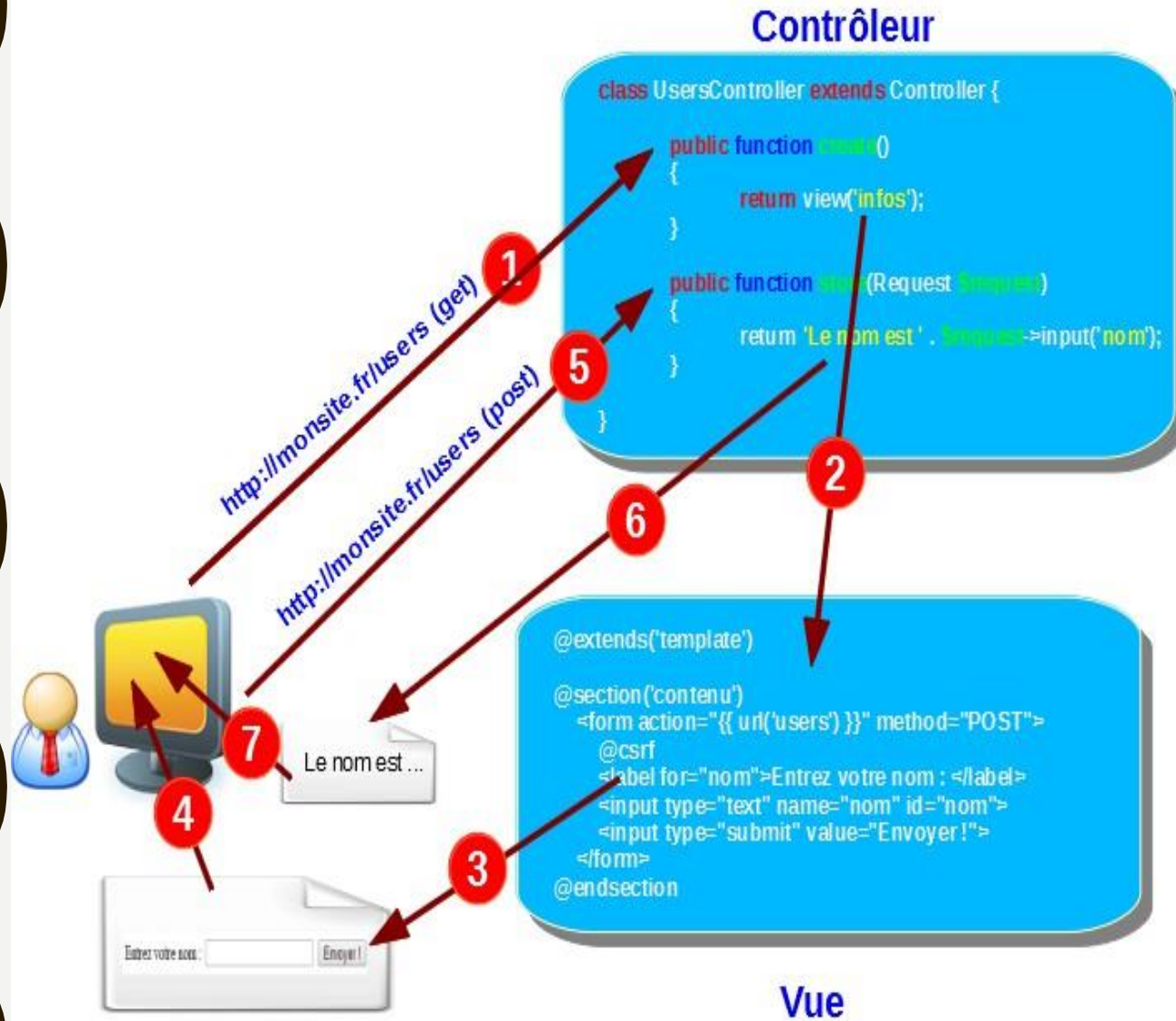
L'helper **url** est utilisé pour générer l'url complète pour l'action du formulaire.  
Pour une route nommée on pourrait utiliser l'helper **route**.

Il ne nous manque plus que le contrôleur pour faire fonctionner tout ça. Utilisez Artisan pour générer un contrôleur :

```
php artisan make:controller UsersController
```

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class UsersController extends Controller
{
    public function create()
    {
        return view('infos');
    }
    public function store(Request $request)
    {
        return 'Le nom est ' . $request->input('nom');
    }
}
```

- la méthode **create** qui reçoit l'url <http://monsite.fr/users> avec le verbe **get** et qui retourne le formulaire,
- la méthode **store** qui reçoit l'url <http://monsite.fr/users> avec le verbe **post** et qui traite les entrées



- (1) le client envoie la requête de demande du formulaire qui est transmise au contrôleur par la route (non représentée sur le schéma),
- (2) le contrôleur crée la vue « infos »,
- (3) la vue « infos » crée le formulaire,
- (4) le formulaire est envoyé au client,
- (5) le client soumet le formulaire, le contrôleur reçoit la requête de soumission par l'intermédiaire de la route (non représentée sur le schéma),
- (6) le contrôleur génère la réponse,
- (7) la réponse est envoyée au client.

# LA PROTECTION CSRF

- Si on regarde le code généré on trouve quelque chose dans ce genre :

```
<input type="hidden" name="_token"  
value="iIjW9PMNsV6VKT2sIc16ShoTf6SdYVZolVUGsxDI">
```

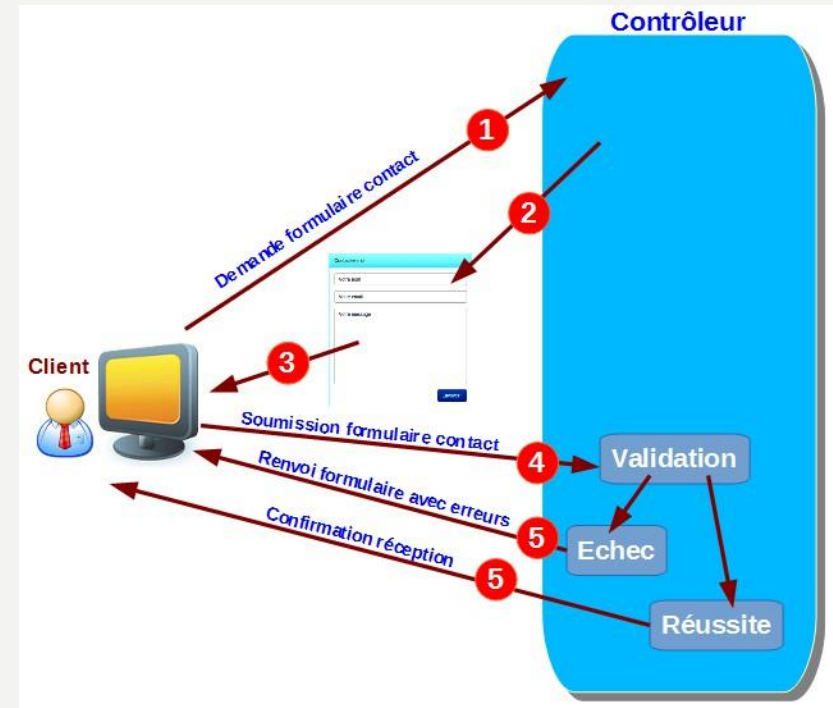
- Tout d'abord **CSRF** signifie **Cross-Site Request Forgery**. C'est une attaque qui consiste à faire envoyer par un client une requête à son insu. Cette attaque est relativement simple à mettre en place et consiste à envoyer à un client authentifié sur un site un script dissimulé (dans une page web ou un email) pour lui faire accomplir une action à son insu.
- Pour se prémunir contre ce genre d'attaque Laravel génère une valeur aléatoire (**token**) associée au formulaire de telle sorte qu'à la soumission cette valeur est vérifiée pour être sûr de l'origine.

# LA VALIDATION

- il est toujours nécessaire de vérifier que ces valeurs correspondent à ce qu'on attend. Par exemple un nom doit comporter uniquement des caractères alphabétiques et avoir une longueur maximale ou minimal, une adresse email doit correspondre à un certain format...

- 1.le client demande le formulaire de contact,
- 2.le contrôleur génère le formulaire,
- 3.le contrôleur envoie le formulaire,
- 4.le client remplit le formulaire et le soumet,
- 5.le contrôleur teste la validité des informations et là on a deux possibilités :

- en cas d'échec on renvoie le formulaire au client en l'informant des erreurs et en conservant ses entrées correctes,
- en cas de réussite on envoie un message de confirmation au client .



```

@extends('templateformulaire')

@section('contenu')
    <form action="{{ url('users') }}" method="POST">
        @csrf
        <label for="nom">Entrez votre nom : </label>
        <input type="text" name="nom" id="nom">
        @error('nom')
        <div>{{ $message }}</div>
        @enderror
        <br/>
        <label for="email">Entrez votre email : </label>
        <input type="text" name="email" id="email">
        @error('email')
        <div>{{ $message }}</div>
        @enderror
        <input type="submit" value="Envoyer !">
    </form>
@endsection

```

- On modifie la vue infos
- En cas de réception du formulaire suite à des erreurs on reçoit une variable **\$errors** qui contient un tableau avec comme clés les noms des contrôles et comme valeurs les textes identifiant les erreurs.
  - *La variable **\$errors** est générée systématiquement pour toutes les vues.*
- On pourrait accéder à cette variable et faire un traitement spécifique pour afficher des erreurs mais Blade nous propose une possibilité plus élégante avec la directive **@error**.
- Enfin en cas d'erreur de validation les anciennes valeurs saisies sont retournée au formulaire et récupérées avec l'helper **old**
-

- effectuer la validation directement dans le contrôleur avec la méthode **validate**

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UsersController extends Controller
{
    public function create()
    {
        return view('infos');
    }

    public function store(Request $request)
    {
        $this->validate($request, [
            'nom' => 'bail|required|between:5,20|alpha',
            'email' => 'bail|required|email'
        ]);
        return 'Le nom est ' . $request-
>input('nom'). ' email est ' . $request->input('email');
    }
}
```

- **Au niveau de validate, elle prend un tableau qui contient des clés correspondent aux champs du formulaire :**

- **bail** : on arrête de vérifier dès qu'une règle n'est pas respectée,
- **required** : une valeur est requise, donc le champ ne doit pas être vide,
- **between** : nombre de caractères entre une valeur minimale et une valeur maximale,
- **alpha** : on n'accepte que les caractères alphabétiques,
- **email** : la valeur doit être une adresse email valide.



# MIGRATIONS ET MODÈLES

# LA CONFIGURATION DE LA BASE

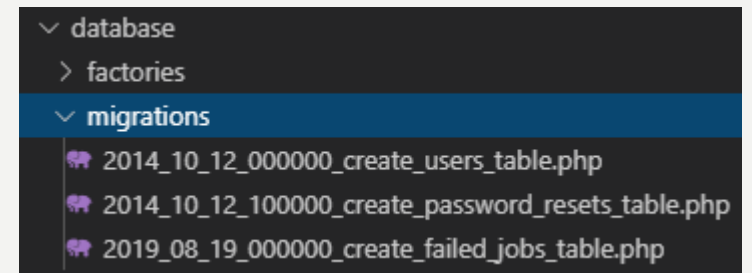
- Vous devez dans un premier temps avoir une base de données. Laravel permet de gérer les bases de type MySQL, Postgres, SQLite et SQL Server. Je ferai tous les exemples avec MySQL mais le code sera aussi valable pour les autres types de bases.
- Il faut indiquer où se trouve votre base, son nom, le nom de l'utilisateur, le mot de passe dans le fichier de configuration **.env** :

```
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=homestead  
DB_USERNAME=homestead  
DB_PASSWORD=secret
```

Ici nous avons les valeurs par défaut à l'installation de Laravel. Il faudra évidemment modifier ces valeurs selon votre contexte de développement et définir surtout le nom de la base, le nom de l'utilisateur et le mot de passe. Pour une installation de MySql en local en général l'utilisateur est **root** et on n'a pas de mot de passe

# MIGRATION

- Une migration permet de créer et de mettre à jour un schéma de base de données. Autrement dit, vous pouvez créer des tables, des colonnes dans ces tables, en supprimer, créer des index... Tout ce qui concerne la maintenance de vos tables peut être pris en charge par cet outil. Vous avez ainsi un suivi de vos modifications
- Si vous regardez dans le dossier **database/migrations** il y a déjà 3 migrations présentes :
  - table **users** : c'est une migration de base pour créer une table des utilisateurs,
  - table **password\_resets** : c'est une migration liée à la précédente qui permet de gérer le renouvellement des mots de passe en toute sécurité,
  - table **failed\_jobs** : une migration qui concerne les queues.



- Commencez par créer une base MySQL et informez **.env**, par exemple :

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

On dispose de 5 commandes pour migrater :

- **fresh** : supprime toutes les tables et relance la migration (commande apparue avec la version 5.5)
- **install** : crée et informe la table de référence des migrations
- **refresh** : réinitialise et relance les migrations
- **rollback** : annule la dernière migration
- **status** : donne des informations sur les migrations

- Lancez alors la commande **install** :

```
php artisan migrate:install
```

- Si vous ouvrez le fichier **database/migrations/2014\_10\_12\_000000\_create\_users\_table.php** vous trouvez
- **up** : ici on a le code de création de la table et de ses colonnes
- **down** : ici on a le code de suppression de la table

- Pour lancer les migrations on utilise la commande **migrate** :

```
php artisan migrate
```

- Pour éviter les erreurs modifier le fichier app/providers/[AppServiceProvider.php](#)

```
<?php

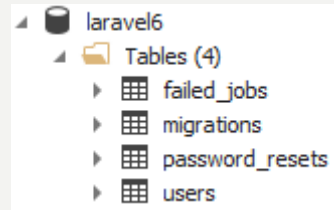
namespace App\Providers;

use Illuminate\Support\Facades\Schema;
use Illuminate\Support\ServiceProvider;

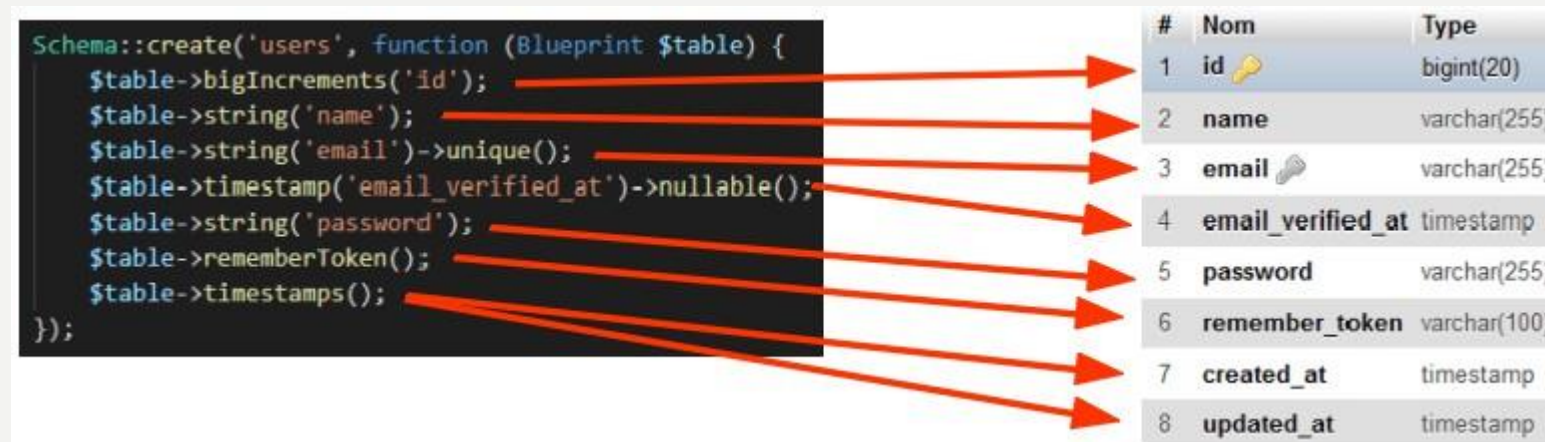
class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Schema::defaultStringLength(191);
    }
}
```

- On voit que les 3 migrations présentes ont été exécutées et on trouve les 3 tables dans la base (en plus de celle de gestion des migrations) :



- Pour comprendre le lien entre migration et création de la table associée voici une illustration pour la table **users** :



# LE MODÈLE ET LA MIGRATION ET ELOQUENT

- Laravel propose un ORM (acronyme de object-relational mapping ou en bon Français un mappage objet-relationnel) très performant.
- Il s'agit que tous les éléments de la base de données ont une représentation sous forme d'objets manipulables.
- Pour simplifier grandement les opérations sur la base de données
- Avec Eloquent une table est représentée par une classe qui étend la classe **Model**. On peut créer un modèle avec Artisan

```
php artisan make:model Test
```

- On peut créer un modèle en même temps que la migration pour la table avec cette syntaxe :

```
php artisan make:model Test -m
```

# PROJET

- Comme exemple pratique et pour voir les section du cour qui manque nous allons prendre le cas d'une table de films
- On repart d'un Laravel vierge et on crée une base comme on l'a vu précédemment. Appelons la par exemple **projet** pour faire original. On renseigne le fichier **.env** en conséquence :  
DB\_CONNECTION=mysql  
DB\_HOST=127.0.0.1  
DB\_PORT=3306  
DB\_DATABASE=projet  
DB\_USERNAME=root  
DB\_PASSWORD=



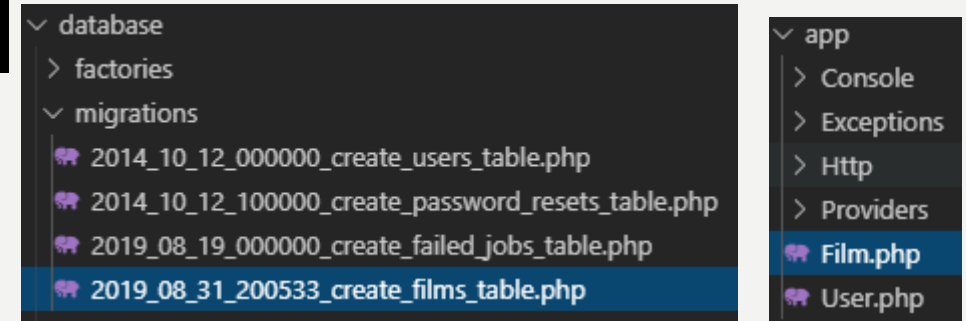
# LA MIGRATION

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

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

- La table possède 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 :

```
php artisan migrate
```

- 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

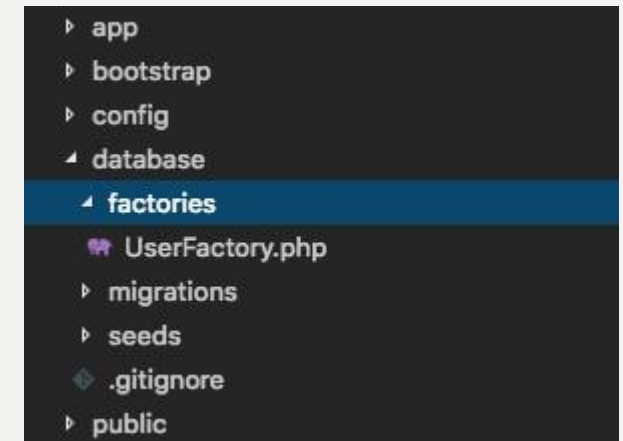
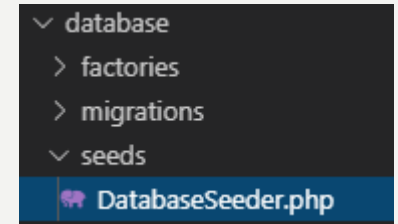
- Le modèle **Film** qu'on a créé est vide au départ. On va se contenter de prévoir l'assignement de masse avec la propriété **\$fillable**

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

- Note :**
- laravel offre un mécanisme pour faciliter l'enregistrement d'informations dans la base de données à partir d'un tableau associatif, par exemple `$request->all()`. On dira qu'on effectue un traitement des champs en lot (tout les champs en une fois). En anglais, on utilisera le terme *mass assignment*.
- Le traitement des champs en lot peut être effectué notamment à l'aide des méthodes **create()**, **fill()** et **update()** ainsi que lors de l'instanciation d'un modèle avec **new**
- Vous pouvez utiliser `fillable` pour protéger les champs que vous souhaitez que cela autorise réellement à la mise à jour.
- Si on ne prend pas quelques précautions, un tel traitement pourrait ouvrir la porte à des vulnérabilités. Par exemple, si on a un champ dans la table des usagers qui donne le statut d'administrateur, un usager malveillant pourrait trafiquer le formulaire de modification d'usager pour lui ajouter un champ qui porte le nom attendu par votre modèle... et lui donner la valeur *true*. Le traitement des champs en lot récupérerait l'ensemble des champs du formulaire et les passerait à la méthode de traitement en lot. Notre usager serait alors promu administrateur

# LA POPULATION

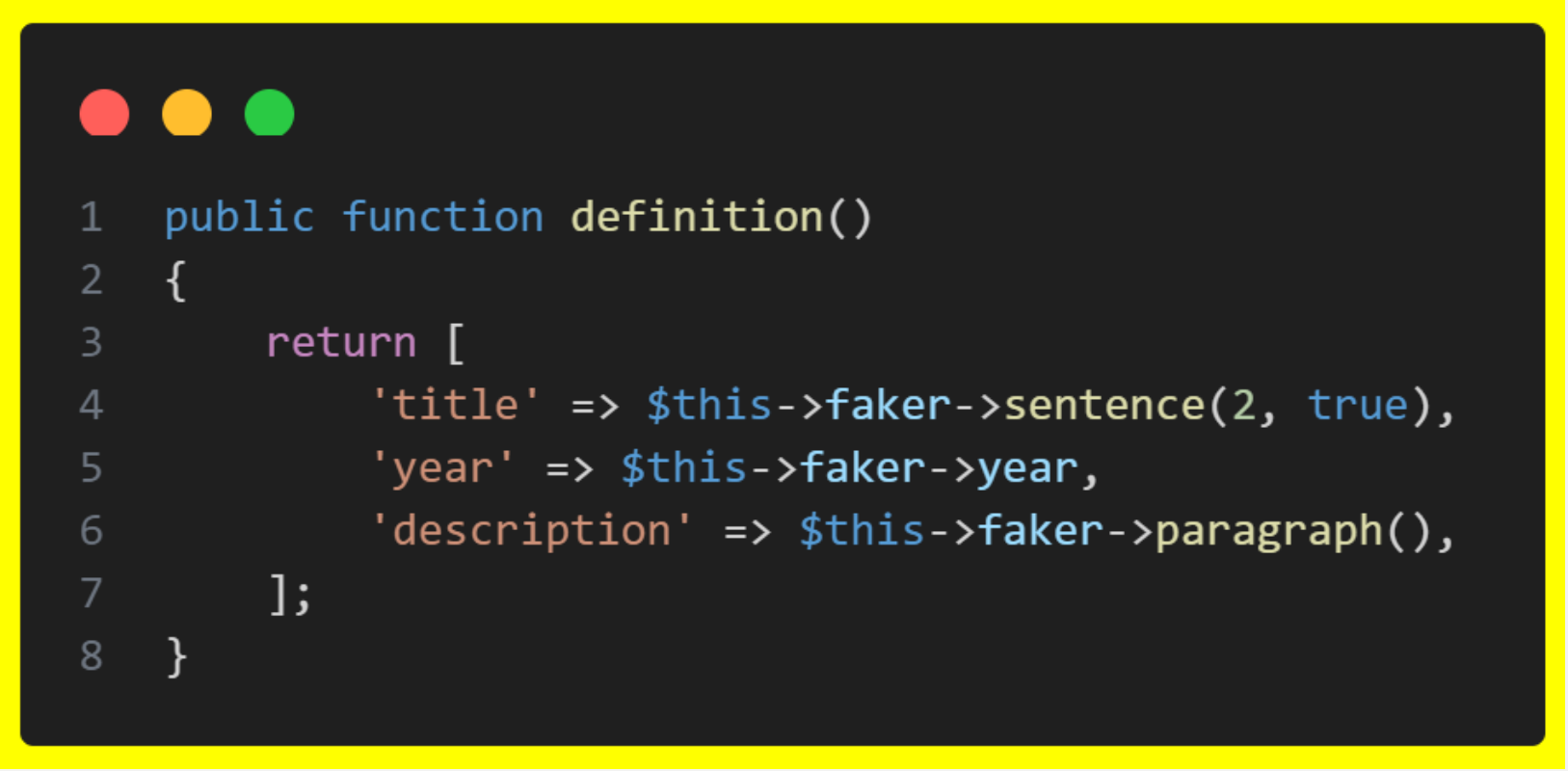
- En plus de proposer des migrations Laravel permet aussi la population (**seeding**), c'est à dire un moyen simple de remplir les tables d'enregistrements. Les classes de la population se trouvent dans le dossier **databases/seeds**,:
- cette simple méthode montre très vite ses limites lorsque l'on veut complexifier nos enregistrements. C'est pourquoi les Factories vont nous être très utiles
- Les Factories ("*usines*" en anglais) sont donc là pour nous permettre de créer des enregistrements en quantité
- Vos Factories se situe dans le dossier **/database/factories**.



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

```
php artisan make:factory FilmFactory
```

On va compléter ainsi :



```
1 public function definition()  
2 {  
3     return [  
4         'title' => $this->faker->sentence(2, true),  
5         'year' => $this->faker->year,  
6         'description' => $this->faker->paragraph(),  
7     ];  
8 }
```

- 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
```

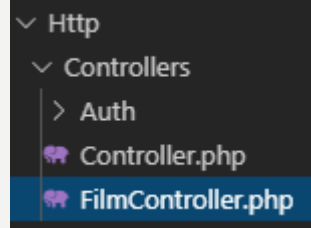
Si tout va bien on se retrouve avec 10 films dans la table :

id	title	year	description
1	Aspernatur quo illo.	1981	Ut tempora nobis et dolorem. 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

- `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** :



- 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::class);
```

- Au lieu de coder plusieurs route on utilise ressource
- On va vérifier ces routes avec Artisan : `php artisan route:list`

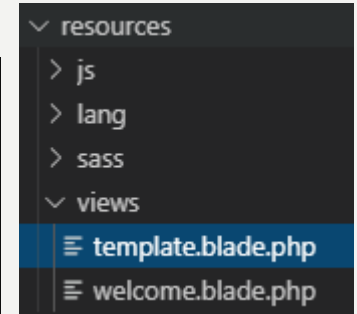
Method	URI	Name	Action	Middleware
GET HEAD	/		Closure	web
GET HEAD	films	films.index	App\Http\Controllers\FilmController@index	web
POST	films	films.store	App\Http\Controllers\FilmController@store	web
GET HEAD	films/create	films.create	App\Http\Controllers\FilmController@create	web
GET HEAD	films/{film}	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.
  - *Le middleware web est automatiquement ajouté à toutes les routes du fichier routes/web.php.*
- 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

# 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 **Bootstrap** 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://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
    @yield('css')
  </head>
  <body>
    <main class="section">
      <div class="container">
        @yield('content')
      </div>
    </main>
  </body>
</html>
```



# LA LISTE DES FILMS

- La liste des films correspond à cette route : `GET|HEAD | films | films.index | App\Http\Controllers\FilmController@index | web`
- 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
- `Film::all();` est équivalent à `select * from film;`
- "compact" est une fonction de php qui permet de créer un tableau automatiquement en prenant les noms des variables

# LA VUE INDEX

On crée la vue **index** avec ce code

views

- index.blade.php
- template.blade.php
- welcome.blade.php

```
@extends('template')
@section('content')
    <div class="card">
        <header class="card-header">
            <h3>Films</h3>
        </header>
        <div class="card-body">

            <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="btn btn-primary" href="{{ route('films.show', $film->id) }}">Voir</a></td>
                            <td><a class="btn btn-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="btn btn-danger" type="submit">Supprimer</button>
                                </form>
                            </td>
                        </tr>
                    @endforeach
                </tbody>
            </table>

        </div>
    </div>
@endsection
```

# L'AFFICHAGE D'UN FILM

- La liste des films correspond à cette route :
- Dans le contrôleur c'est la méthode **show** qui est concernée. On va donc la coder.
- 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)
```

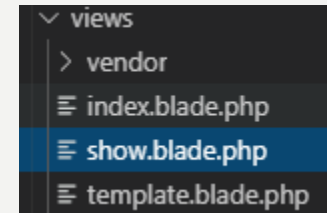
- La variable **id** contient la valeur passée dans l'url. Par exemple **.../films/8** indique qu'on veut voir les informations du film d'identifiant 8. Il suffit donc ensuite d'aller chercher dans la base le film correspondant.
- On va utiliser une autre stratégie L'argument cette fois est une instance 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">
            <h4>Titre : {{ $film->title }}</h4>
        </header>
        <div class="card-body">
            <p>Année de sortie : {{ $film->year }}</p>
            <hr>
            <p>{{ $film->description }}</p>
        </div>
    </div>
@endsection
```



**Titre : Aspernatur quo illo.**

Année de sortie : 1981

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

# SUPPRIMER UN FILM

- La suppression d'un film correspond à cette route :

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

- 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.
- Si on voulait utiliser les eloquent on pouvez passer le \$id comme paramètre

```
public function destroy($id)
{
    Film::find($id)->delete();
    return back()->with('info', 'Le film a bien été supprimé dans la base de données.');
```

\* Find ici est une methode qui recherche le id

- 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="alert alert-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



# CRÉER UN FILM

- Pour la création d'un film on va avoir deux routes :
- pour afficher le formulaire de création
- pour soumettre le formulaire

# CRÉER UN FILM

- Pour la création d'un film on va avoir deux routes :
- pour afficher le formulaire de création
- pour soumettre le formulaire

POST	films	films.store	App\Http\Controllers\FilmController@store	web
GET HEAD	films/create	films.create	App\Http\Controllers\FilmController@create	web

- Dans le contrôleur ce sont les méthodes **create** et **store** qui sont concernées. On va donc les coder :

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

public function store(Request $request)
{
    $this->validate($request, [
        'title' => ['required', 'string', 'max:100'],
        'year' => ['required', 'numeric', 'min:1950', 'max:' . date('Y')],
        'description' => ['required', 'string', 'max:500'],
    ]);
    Film::create($request->all());
    return redirect()->route('films.index')->with('info', 'Le film a bien été créé');
}
```

Model::create est lequivalent de :insert into table ,,

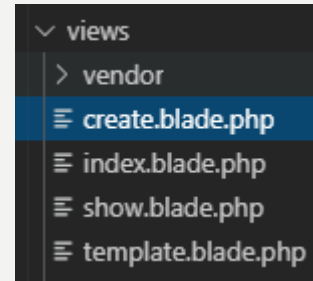
- On a déjà codé la vue `index` mais on n'a pas prévu de bouton pour créer un film on complete notre code comme suit.

```
@section('css')
<style>
  .card-footer {
    justify-content: center;
    align-items: center;
    padding: 0.4em;
  }
  .is-info {
    margin: 0.3em;
  }
</style>
@endsection
@section('content')
@if(session()->has('info'))
<div class="alert alert-success">
  {{ session('info') }}
</div>
@endif
<div class="card">
  <header class="card-header">
    <h3>Films</h3>
    <a class="btn btn-
info" href="{{ route('films.create') }}">Créer un film</a>
  </header>
```

... \*

# LA VUE CREATE

- On crée la vue create :



```
@extends('template')
@section('content')
    <div class="card">
        <header class="card-header">
            <h4>Création d'un film</h4>
        </header>
        <div class="card-body">

            <form action="{{ route('films.store') }}" method="POST">
                @csrf
                <div class="field">
                    <label class="label">Titre</label>
                    <div class="control">
                        <input class="form-control" @error('title') is-
danger @enderror" type="text" name="title" value="{{ old('title') }}" placeholder="Titre du film">
                    </div>
                    @error('title')
                        <p class="help is-danger">{{ $message }}</p>
                    @enderror
                </div>
            </form>
        </div>
    </div>
```



# MODIFIER UN FILM

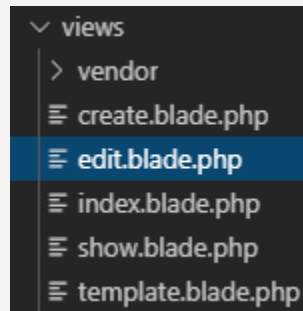
- Pour la modification d'un film on va avoir deux routes :
- pour afficher le formulaire de modification
- pour soumettre le formulaire
- Dans le contrôleur ce sont les méthodes **edit** et **update** qui sont concernées. On va donc les coder :

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

```
public function edit(Film $film)
{
    return view('edit', compact('film'));
}
public function update(Request $filmRequest, Film $film)
{
    $film->update($filmRequest->all());
    return redirect()->route('films.index')->with('info', 'Le film a bien été modifié');
}
```

# LA VUE EDIT

- Là aussi on va avoir pratiquement le même code que la vue **create**. La différence c'est qu'il faut renseigner les contrôles du formulaire au départ. On crée la vue **edit** :





```

...
    <form action="{{ route('films.update',$film->id) }}" method="POST">
        @csrf
        @method('put')
    ...
        <input class="form-control @error('title') is-
danger @enderror" type="text" name="title" value="{{ old('title',$film->title) }}" placeholder="Titre du film">
    ...
        <input class="form-control" type="number" name="year" value="{{ old('year', $film-
>year) }}" min="1950" max="{{ date('Y') }}">
    </div>
...
        <textarea class="form-
control" name="description" placeholder="Description du film">{{ old('description', $film->description) }}</textarea>
    ...

```

# RELATION 1:N

- 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...

# 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 -m
```

- 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** :

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

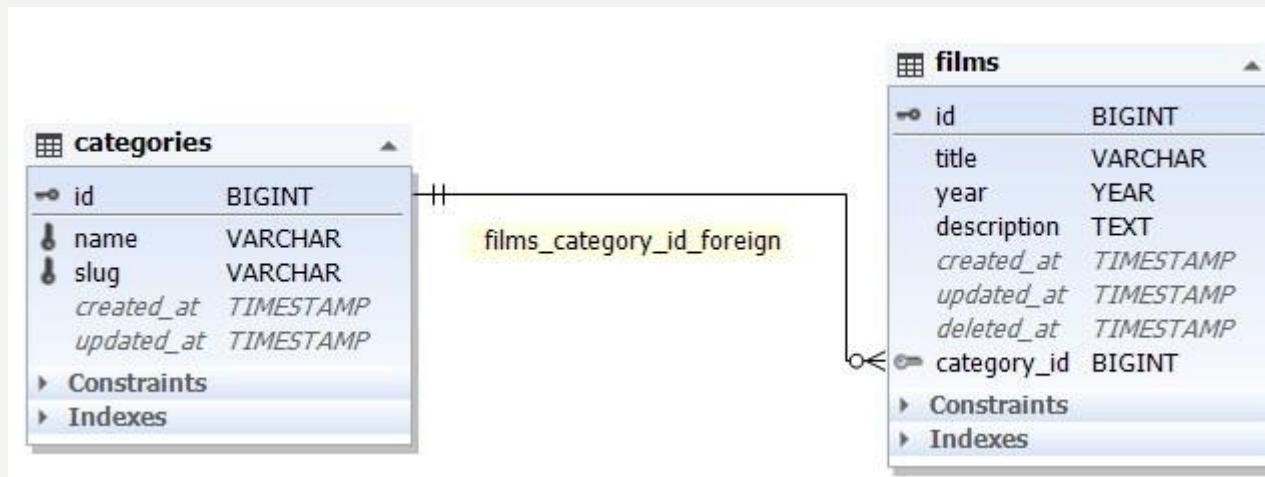
Dans la table **films** on déclare une clé étrangère (**foreign**) nommée **category\_id** qui référence (**references**) la colonne **id** dans la table (**on**) **categories**. En cas de suppression (**onDelete**) ou de modification (**onUpdate**) on a une restriction (**restrict**)

- On va lancer les migrations en rafraichissant la base

```
php artisan migrate:fresh
```

# LA RELATION

- On a la situation suivante :
- une catégorie peut avoir plusieurs films,
- un film n'appartient qu'à une catégorie.



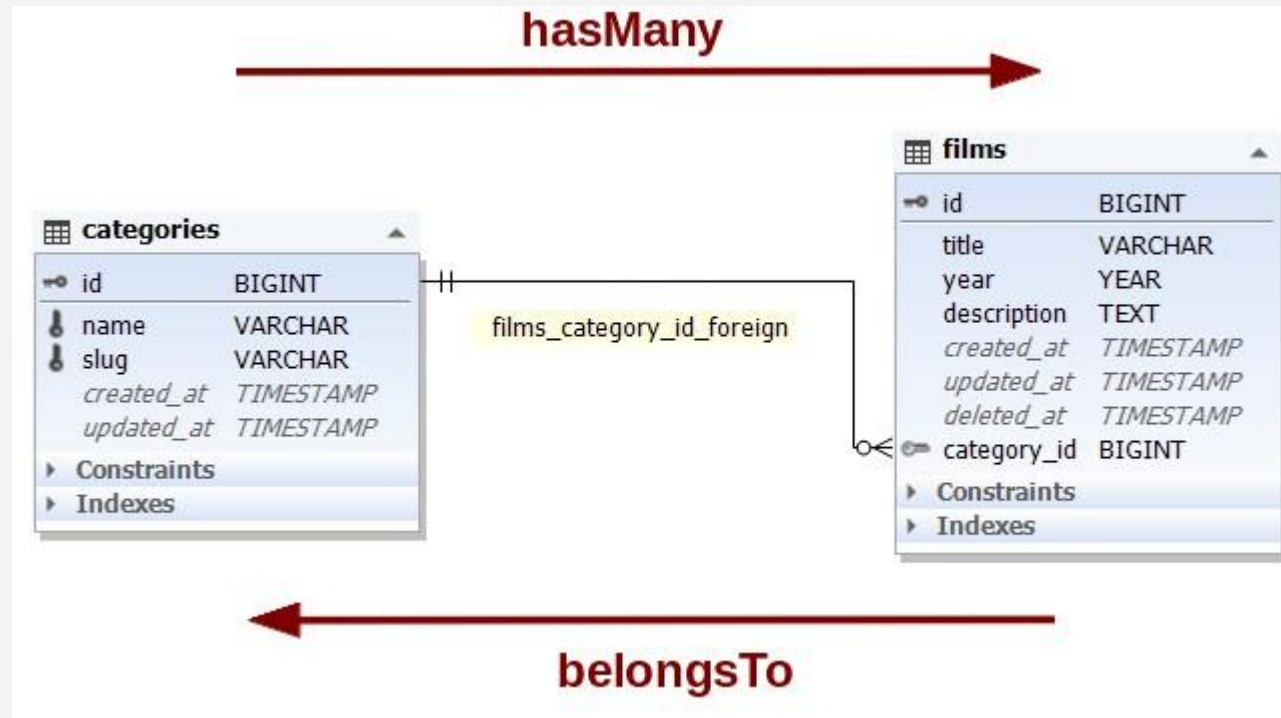
# LE MODÈLE CATEGORY / FILM

```
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

```
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 CRÉATION D'UN FILM

```
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>
```



# ROUTE ET CONTRÔLEUR

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

```
public function index($slug=null)
{
    $query = $slug ? Category::whereSlug($slug)->firstOrFail()->films()->get() : Film::all();

    $films = $query;
    $categories = Category::all();

    return view('index', compact('films', 'categories', 'slug'));
}
```

# LA VUE INDEX

```
<h3>Films</h3>
  <div class="select">
    <select onchange="window.location.href = this.value">
      <option value="{{ route('films.index') }}" @unless($slug) selected @endunless>Toutes caté
gories</option>
      @foreach($categories as $category)
        <option value="{{ route('films.category', $category->slug) }}" {{ $slug == $category-
>slug ? 'selected' : '' }}>{{ $category->name }}</option>
      @endforeach
    </select>
  </div>
  <a class="btn btn-info" href="{{ route('films.create') }}">Créer un film</a>
```

# LA VUE SHOW

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

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

# L'AUTHTIFICATION

- Dans l'installation de base vous ne trouvez aucune route pour l'authentification. Pour les créer (et ça ne créera pas seulement les routes) il faut déjà installer un package :

```
composer require laravel/ui
```

```
php artisan ui bootstrap --auth
```

```
npm install  
npm run dev
```

il faut disposer de node pour  
que ça marche

```
php artisan migrate
```

- Une fois installer des route sont ajouter dans le fichier web



# ANPT