

TD Mangas World

Introduction

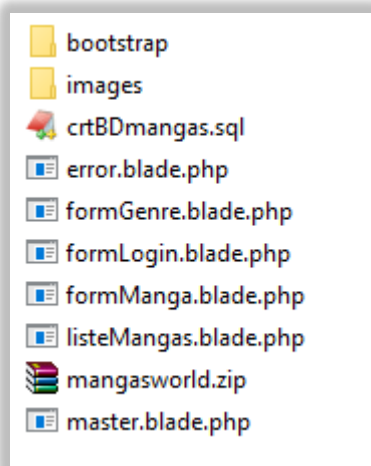
Pour illustrer la mise en œuvre d'un framework orienté MVC, nous allons développer une (petite) application web de présentation de bandes dessinées de type Mangas sous le framework Laravel. Pour ce faire vous disposerez d'un certain nombre de ressources qui sont situées dans le dossier Ressources.



Copier le fichier MangasWorld.zip dans votre espace personnel.

Attention : le fichier MangasWorld.zip contient des ressources différentes qui seront utilisées tout au long de ce TD. Il faut décompresser ce fichier et cette action de décompression est particulièrement longue si elle se fait à travers le réseau, il convient donc de procéder de la manière suivante :

- Copier le fichier MangasWorld.zip sur un disque local, par exemple C:\Temp,
- Décompresser ce fichier dans ce même dossier pour obtenir la liste suivante :



Effectuer les opérations demandées.

Préparation de l'environnement

PHP MySQL

Nous allons avoir besoin d'un environnement PH et MySQL, pour cela une des meilleures solutions actuellement c'est xampp : <https://www.apachefriends.org/fr/index.html>.

Normalement xampp est déjà installé, si ce n'est pas le cas, l'installer sur le disque Data.

Important : depuis la version 5.4 Laravel utilise PHP7, c'est donc la version php7 d'xampp qu'il faut installer.

Composer

Composer est un gestionnaire de dépendances qui va nous permettre de télécharger de façon automatisée l'ensemble des ressources nécessaires au développement d'une application web sous Laravel. Il y a deux façons d'installer Composer.

En tant qu'administrateur

Si on a les droits administrateur sur sa machine, on télécharge le setup depuis l'adresse suivante : <https://getcomposer.org/Composer-Setup.exe> et on l'installe (conserver les options par défaut).

On peut vérifier son installation en lançant dans une console : `composer -V`

En tant qu'utilisateur

L'installation se fait en mode console, il faut descendre dans le dossier xampp\htdocs, puis lancer les lignes suivantes :

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') ===
'55d6eade61b29c7bdee5cccfb50076874187bd9f21f65d8991d46ec5cc90518f447387fb9f76eae
1fbbacf329e583e30') { echo 'Installer verified'; } else { echo 'Installer
corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Lignes que l'on peut récupérer à l'adresse suivante : <https://getcomposer.org/download/>

On crée ensuite une commande composer.bat comme suit :

```
echo @php "%~dp0composer.phar" %*>composer.bat
```

Commande que l'on peut trouver à l'adresse suivante : <https://getcomposer.org/doc/00-intro.md#manual-installation>

On peut vérifier son installation en lançant dans une console : `composer -V`

Installateur Laravel

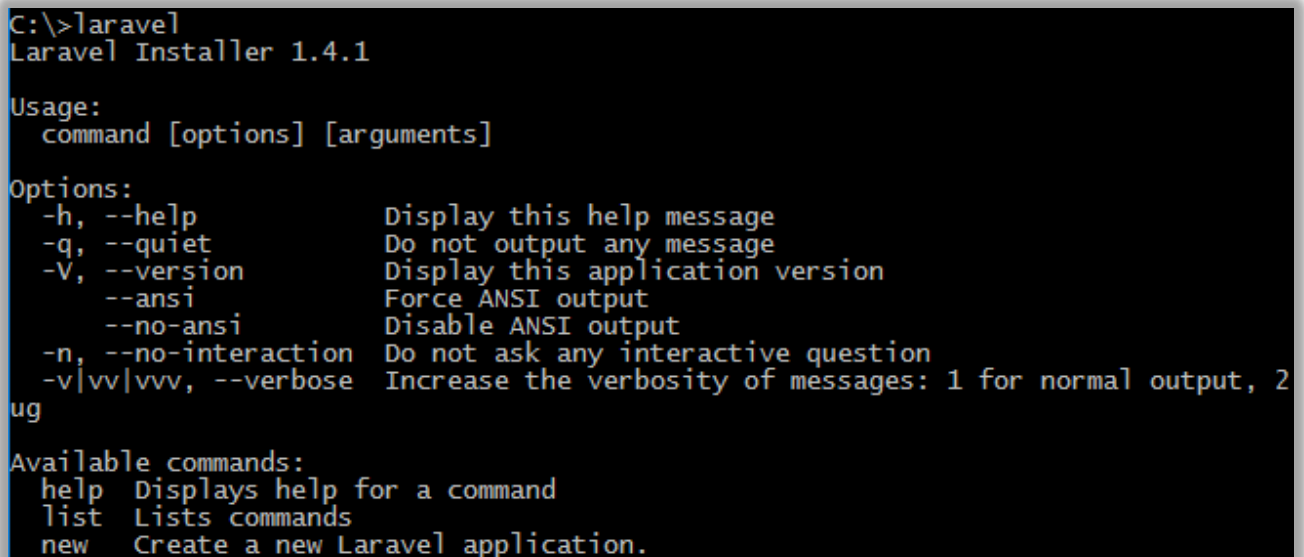
Le framework Laravel dispose d'un installateur qui utilise directement composer, si on désire utiliser cet installateur (ce n'est pas une obligation), il faut l'installer en global ou en local.

Administrateur

En tant qu'administrateur, l'installation se fera en global :

```
composer global require "laravel/installer"
```

On peut vérifier que cela fonctionne correctement en tapant la commande `laravel` :



```
C:\>laravel
Laravel Installer 1.4.1

Usage:
  command [options] [arguments]

Options:
  -h, --help                Display this help message
  -q, --quiet               Do not output any message
  -V, --version             Display this application version
  --ansi                   Force ANSI output
  --no-ansi                Disable ANSI output
  -n, --no-interaction      Do not ask any interactive question
  -v|vv|vvv, --verbose     Increase the verbosity of messages: 1 for normal output, 2
ug

Available commands:
  help  Displays help for a command
  list  Lists commands
  new   Create a new Laravel application.
```

Utilisateur

L'installation se fera en local, de préférence dans le dossier `c:\xampp\htdocs` :

```
composer require "laravel/installer"
```

Il faut créer son propre lanceur comme suis, sachant qu'il ne sera actif que dans le dossier dans lequel il a été créé, ici `c:\xampp\htdocs` :

```
echo c:\xampp\htdocs\vendor\bin\laravel > laravel.bat
```

On peut vérifier que cela fonctionne correctement en tapant la commande suivante :

```
laravel
```

```
C:\xampp\htdocs>laravel
Laravel Installer version 1.3.3

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi        Disable ANSI output
  -n, --no-interaction Do not ask any interactive question
  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands:
  help  Displays help for a command
  list  Lists commands
  new   Create a new Laravel application.
```

Création du projet

Le projet

Il y a deux façons de créer le projet, en le créant ex nihilo ou en récupérant un pré-projet dans les Ressources.

Important : comme la plupart des postes de travail seront partagés par les deux groupes, il est plus que prudent de créer un dossier portant son nom dans le dossier c:\xampp\htdocs et d'y créer ses projets. Pour qu'il n'y ait pas d'ambiguïté, le support fera référence au dossier arsane, à vous de transposer dans vos actions.



Réaliser l'opération demandée.

Remarque : pensez à bien faire une copie de votre projet avant de quitter la salle en fin de séance.

Ex nihilo sans l'installateur



Dans une console, sous xampp\htdocs saisir la commande suivante :

```
composer create-project --prefer-dist laravel/laravel arsane/mangasworld
```

Note : cela peut prendre un certain temps, c'est surtout fonction de la bande passante dont on dispose sur internet.

Ex nihilo avec l'installateur

```
laravel new arsane/mangasworld
```

Il faut ensuite configurer le projet pour qu'il puisse gérer les formulaires.



Ajouter dans composer.json la référence à la bibliothèque de formulaires (voir ci-dessous) et modifier les sections providers et aliases du fichier config/app.php (voir ci-dessus).

Fichier composer.json

```
"require": {
    "php": ">=7.0.0",
    "fideloper/proxy": "~3.3",
    "laravel/framework": "5.5.*",
    "laravel/tinker": "~1.0",
    "laravelcollective/html": "5.5.*"
},
```

Fichier config/app.php

Rajouter dans la partie providers la ligne en surbrillance

```
Illuminate\Session\SessionServiceProvider::class,  
Illuminate\Translation\TranslationServiceProvider::class,  
Illuminate\Validation\ValidationServiceProvider::class,  
Illuminate\View\ViewServiceProvider::class,  
Collective\Html\HtmlServiceProvider::class,  
  
/*  
 * Application Service Providers...  
*/
```

Rajoutez dans le tableau de la partie aliases les lignes Form et Html

```
'URL' => Illuminate\Support\Facades\URL::class,  
'Validator' => Illuminate\Support\Facades\Validator::class,  
'View' => Illuminate\Support\Facades\View::class,  
'Form' => 'Collective\Html\FormFacade',  
'Html' => 'Collective\Html\HtmlFacade',  
  
],  
];
```



Descendre dans le dossier mangasworld et mettre à jour le framework en lançant la commande : `..\composer update` ou `composer update` selon le type d'installation.

```
C:\xampp\htdocs\arsane\myProject>composer update  
Loading composer repositories with package information  
Updating dependencies (including require-dev)  
Package operations: 1 install, 1 update, 0 removals  
- Updating doctrine/instantiator (1.0.5 => 1.1.0): Downloading (100%)  
- Installing laravelcollective/html (v5.5.1): Downloading (100%)  
Writing lock file  
Generating optimized autoload files  
> Illuminate\Foundation\ComposerScripts::postAutoloadDump  
> @php artisan package:discover  
Discovered Package: fideloper/proxy  
Discovered Package: laravel/tinker  
Discovered Package: laravelcollective/html  
Package manifest generated successfully.
```

A partir du pré-projet

L'ensemble des opérations décrites ci-dessus ont été déjà réalisées, il suffit de décompresser le fichier mangasworld.zip (tout en minuscules) dans le dossier xampp\htdocs\arsane.

Le projet NetBeans

Nous allons maintenant intégrer notre application Laravel dans un projet NetBeans.

Si on a créé soi-même le projet, procéder comme suit :



Lancer NetBeans, New Project, PHP, PHP Application with Existing Sources,
Sources Folder : c:\xampp\htdocs\arsane\mangasworld,
PHP Version : la plus haute, Next,
Project URL : <http://localhost/arsane/mangasworld/public>,
Finish,
Lancer le Control panel de Xampp, et dans Netbeans cliquer sur Run (triangle vert) pour vérifier que le projet se lance correctement.

TD Mangas World

Si on a dézippé le projet de base fourni dans les Ressources, procéder comme suit :



Lancer netBeans, Open Project, descendre dans le dossier htdocs\votre_nom et sélectionner le projet mangasworld. Faire un clic-droit sur le nom du projet, sélectionner Run Configuration et remplacer arsane dans le Project URL, par son propre nom.

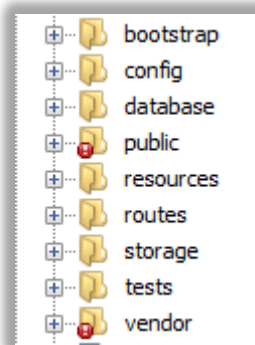
Structure d'une application Laravel

Quelle que soit la méthode utilisée pour installer un projet, on aboutit à une structure assez complexe dont voici les principaux éléments dans le dossier app :

- **Console** : ici on mettra toutes les commandes en mode console, il y a au départ une commande Inspire qui sert d'exemple,
- **Exceptions** : ici on mettra ici des gestionnaires d'exceptions,
- **Http** : ici on va trouver tout ce qui concerne la communication : contrôleurs, modèles, routes, middlewares (il y a 4 middlewares de base) et requêtes HTTP,
- **Providers** : ici on va mettre tous les providers, il y en a déjà 4 au départ. Les providers servent à initialiser les composants.

On trouve également le fichier User.php qui est un modèle qui concerne les utilisateurs pour la base de données. Nous n'utiliserons pas cette méthode qui est très complexe à comprendre et à mettre en œuvre.

Autres dossiers



Voici une description du contenu des 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...
- **vendor** : tous les composants de Laravel et de ses dépendances,
- **config** : toutes les configurations : application, authentification, cache, base de données, espaces de noms, emails, systèmes de fichier, session...
- **database** : ici on aura les migrations et les populations,
- **resources** : ici on aura les vues (dossier views), les fichiers de langage et les assets (par exemple les fichiers LESS ou Sass),
- **routes** : c'est ici que se fera la gestion des routes,
- **storage** : pour stocker les données temporaires de l'application : vues compilées, caches, clés de session...
- **tests** : pour les fichiers de tests unitaires,
- **vendor** : ce dossier contient le framework et ses extensions.

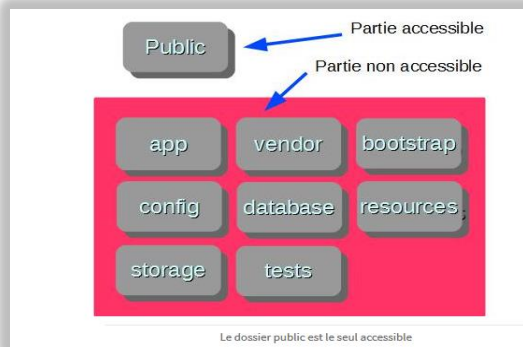
Fichiers de la racine

Il y a un certain nombre de fichiers dans la racine dont voici les principaux :

- **artisan** : outil en ligne de Laravel pour des tâches de gestion,
- **composer.json** : fichier de référence de Composer, fichier très important car c'est lui qui va déterminer les composants et leurs dépendances qui vont être utilisés dans l'application,
- **phpunit.xml** : fichier de configuration de phpunit (pour les tests unitaires),
- **.env** : fichier pour spécifier l'environnement d'exécution et notamment les codes d'accès à la base de données.

Accessibilité

Pour des raisons de sécurité sur le serveur seul le dossier Public doit être accessible :



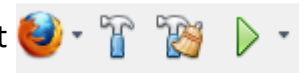
Tester le projet



Lancer XAMPP (Apache et MySQL), puis cliquer sur le triangle vert

Ou bien dans un navigateur saisir l'url suivante :

http://localhost/votre_nom/mangasworld/public/



La base de données



Sous phpmyadmin, créer la base de données mangas à partir du script crtBDMangas.sql se trouvant dans Ressources. Modifier le fichier .env avec les données suivantes :

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

L'IHM

Bootstrap

L'Interface Homme/ Machine va être basée sur bootstrap qui est le standard de configuration css des sites web. Nous allons donc installer le framework bootstrap et le référencer dans la page principale constituée par le fichier master.blade.php.



Dans le dossier public, créer le dossier assets et dans ce dossier copier les dossiers css, fonts et js se trouvant dans le dossier Ressources.

Note : il est possible de faire un copier sous Windows Explorer et de faire un coller sous NetBeans.

La page principale

Nous allons utiliser les fonctionnalités du moteur de Templates Blade. Donc nous n'adresserons que des vues qui s'inséreront dans la page principale constituée par le fichier master.blade.php. Cette page est complexe à réaliser et cela demanderait trop de temps et d'énergie que de la construire ex nihilo, vous allez donc le récupérer dans Ressources et la compléter.

Référencement de bootstrap

Le référencement du framework bootstrap se fait à deux endroits, dans la partie HEAD pour les fichiers css et juste avant le </body> pour les fichiers javascript. Ce référencement utilise la syntaxe Blade comme suit :

Fichiers css

```
{!! Html::style('assets/css/bootstrap.css') !!}
{!! Html::style('assets/css/mangas.css') !!}
```

Fichiers javascript

```
{!! Html::script('assets/js/bootstrap.min.js') !!}
{!! Html::script('assets/js/ui-bootstrap-tpls.js') !!}
{!! Html::script('assets/js/bootstrap.js') !!}
```

L'emplacement des vues

L'emplacement où viendront s'insérer les vues se définit à l'aide de l'instruction Blade @yield('nom_emplacement'). Il est possible de définir plusieurs emplacements au sein de la même page, mais pour le moment nous n'en définirons qu'un seul.

```
</nav>
</div>
<div class="container">
    @yield('content')
</div>
```

Réalisation



Dans le dossier views, créer le dossier layouts, copier dans ce dossier le fichier master.blade.php pris dans Ressources et remplacer les annotations /* A compléter ... */ par les éléments requis présentés ci-dessus.

La route par défaut

Lorsqu'on appelle le site par l'url de base (http://localhost/mangasworld/public/) on fait appel à la route par défaut qui pour le moment appelle la page welcome.blade.php. Nous allons remplacer cette page par la vue [home.blade.php](#) qui viendra s'insérer à l'emplacement défini par l'instruction @yield().



Dans le dossier views, supprimer la page welcome, créer la vue [home.blade.php](#) (penser à supprimer la balise d'ouverture php <?php) et saisir le code ci-dessous :

```
@extends('layouts.master')
@section('content')
<div>
    <h1 class="bvn"> Bienvenue sur Mangas World ! </h1>
</div>
@stop
```

@extends indique à quelle page on fait référence, @section indique où va s'insérer la vue et @stop quand s'arrête cette insertion.

La gestion des routes

Lorsque dans l'application on cliquera sur un lien ou sur un bouton, on enverra au serveur http une requête qui sera interceptée par le routeur. Ici ce sera le fichier routes/web.php qui sera sollicité pour réagir à la requête. Ce fichier est donc central dans l'architecture Laravel.

Le routage réagit aux différents verbes http : essentiellement GET, POST, mais aussi PUT, DELETE, OPTIONS et PATCH. Une route peut rediriger vers une Vue :

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

Ou vers une méthode d'un contrôleur :

```
Route::get('/login', 'UtilisateurController@login');
```

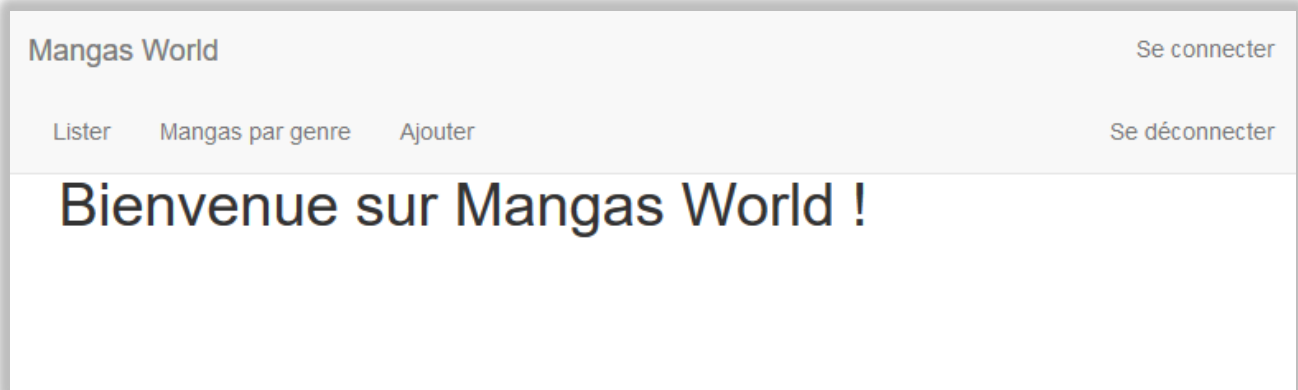
Modification du routage par défaut

Par défaut, nous ne solliciterons plus la vue welcome, mais la vue home, donc dans le fichier routes/web.php, il faudra modifier la route par défaut comme suit :

```
// Route par défaut
Route::get('/', function () {
    return view('home');
});
```




Effectuer les modifications demandées et tester, normalement le navigateur devrait afficher ceci :



Note : nous allons rectifier le petit problème du menu général dans le prochain chapitre !

L'authentification

Gestion des items de menu

Avant de pouvoir accéder aux différentes fonctionnalités, il faut s'être authentifié. Il y a plusieurs façons de gérer cette authentification, le choix qui a été fait ici et d'opter pour une solution simple facilement mise en œuvre. Les items de menus donnant accès aux différentes fonctionnalités ne seront accessibles que si l'utilisateur s'est authentifié (login et mot de passe). Pour se faire nous utiliserons deux éléments : la Session et les structures de contrôle de Blade, le moteur de Template.

Lorsque l'utilisateur se sera authentifié, on stockera son id dans la Session. Donc si un id est présent dans la Session, il aura accès aux fonctionnalités (Lister, Ajouter ...) à contrario il n'aura accès qu'à la possibilité de se connecter.

Voici la nouvelle version du menu :

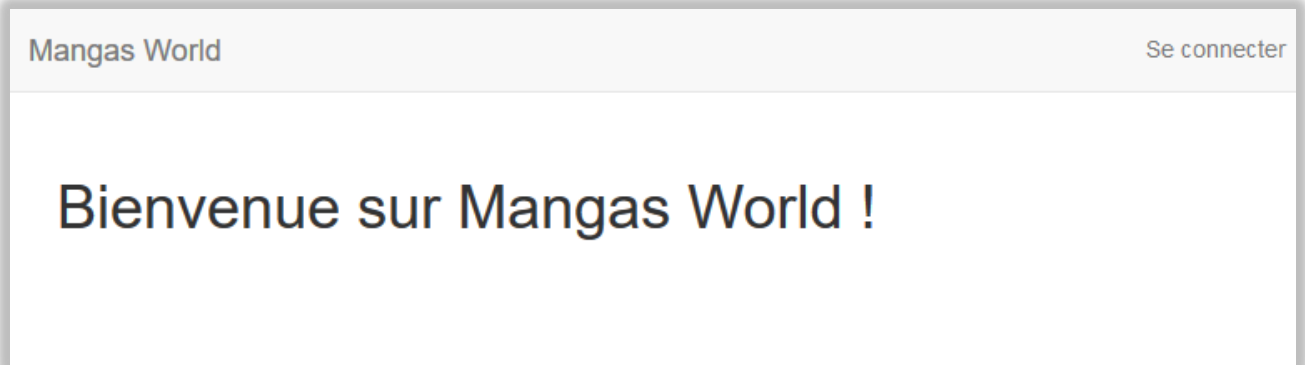
```
</div>
@if (Session::get('id') == 0)
<div class="collapse navbar-collapse" id="navbar-collapse-target">
  <ul class="nav navbar-nav navbar-right">
    <li><a href="{{ url('/getLogin') }}" data-toggle="collapse" data-target="#navbar-collapse-target">Se connecter</a></li>
  </ul>
</div>
@endif
@if (Session::get('id') > 0)
<div class="collapse navbar-collapse" id="navbar-collapse-target">
  <ul class="nav navbar-nav">
    <li><a href="{{ url('/listerMangas') }}" data-toggle="collapse" data-target="#navbar-collapse-target">Lister</a></li>
    <li><a href="{{ url('/listerGenres') }}" data-toggle="collapse" data-target="#navbar-collapse-target">Mangas par genre</a></li>
    <li><a href="{{ url('/ajouterManga') }}" data-toggle="collapse" data-target="#navbar-collapse-target">Ajouter</a></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li><a href="{{ url('/signOut') }}" data-toggle="collapse" data-target="#navbar-collapse-target">Se déconnecter</a></li>
  </ul>
</div>
@endif
```

TD Mangas World

On voit ici tout l'intérêt d'un moteur de Template (Blade en l'occurrence, mais les autres disposent des mêmes fonctionnalités), il peut accéder aux données générées par le framework.



Modifier master.blade.php et tester, normalement on devrait obtenir ceci :



On pourra consulter le code source de la page (Ctrl + u) et constater que seul l'item se connecter a été généré. Cela permet d'éviter qu'un utilisateur averti n'ait accès aux fonctionnalités non permises en consultant le code source de la page.

Se connecter

Pour permettre à un utilisateur de se connecter, nous aurons besoin des composants suivants :

Composant	Valeur	Rôle
Route	/getLogin	Prise en compte du clic sur l'item Liste du menu
Vues	formLogin.blade.php error.blade.php	Formulaire de login Affichage des messages d'erreurs
Contrôleur	UtilisateurController	Pilotage de la demande de connexion

La vue formLogin

C'est un formulaire classique de connexion, il faudra lui ajouter les commandes Blade qui lui permettront de s'insérer dans la page master.

```
@extends('layouts.master')
@section('content')
{!! Form::open(['url' => 'signIn']) !!}
<div class="col-md-12 well well-md">
    <center><h1>Authentification</h1></center>
    <div class="form-horizontal">
        <div class="form-group">
            <label class="col-md-3 control-label">Identifiant : </label>
            <div class="col-md-6 col-md-3">
                <input type="text" name="login" class="form-control" placeholder="Identifiant" />
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-3 control-label">Mot de passe : </label>
            <div class="col-md-6 col-md-3">
                <input type="password" name="pwd" class="form-control" placeholder="Mot de passe" />
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-6 col-md-offset-3">
                <button type="submit" class="btn btn-default btn-primary">Valider</button>
            </div>
            <div class="col-md-6 col-md-offset-3">
                @include('error')
            </div>
        </div>
    </div>
</div>
{!! Form::close() !!}
@stop
```

Lorsque l'utilisateur clique sur le bouton valider, on appellera la route /signIn. On remarquera aussi que l'on inclut la vue error.blade.php en bas du formulaire. Nous traiterons ces deux points dans le chapitre suivant.



Copier les deux vues formLogin et error et effectuer les modifications demandées.

Le contrôleur UtilisateurController

Nous allons avoir besoin d'un contrôleur pour piloter l'authentification, ce sera le UtilisateurController. Pour le moment son rôle se limitera à afficher la vue formLogin.

Pour créer un contrôleur, il faut repasser sur une console, descendre dans le dossier racine de l'application et saisir le code suivant : **php artisan make:controller NomController**

Par défaut, les contrôleurs sont créés dans le dossier app/http/Controllers, ils héritent de la classe Controller et doivent être post-fixés du terme Controller.



Créer le contrôleur UtilisateurController et le compléter comme suit :

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
class UtilisateurController extends Controller
{
    /**
     * Initialise le formulaire d'authentification
     * @return Vue formLogin
     */
    public function getLogin() {
        $erreur = "";
        return view('formLogin', compact('erreur'));
    }
}
```

La vue error qui va s'insérer dans le formulaire a pour rôle d'afficher les messages d'erreurs qui seront placés dans la variable \$erreur qui pour le moment est initialisée à vide.

La fonction php compact('var1', 'var2', ...) transforme une liste de variables dont on fournit le nom en un tableau associatif dont la clé est le nom de la variable et le contenu sa valeur. Ici le tableau est réduit à une seule occurrence ('erreur'), mais plus tard nous pourrions en avoir deux, trois Ce tableau est ensuite passé à la vue qui pourra l'utiliser grâce aux structures de contrôle du moteur de Template Blade.

La vue error

```
@if ($erreur!= "")
<p>
    <div class="alert-danger" role="alert">
        <span class="glyphicon glyphicon-exclamation-sign"
            aria-hidden="true"></span> {{ $erreur or '' }}
    </div>
</p>
@endif
```

Si la variable \$erreur est non vide alors on affiche la balise sous-jacente. On a pris la précaution d'afficher une chaîne vide si la variable \$erreur n'était pas définie : {{ \$erreur or '' }}. Ce n'était pas indispensable ici, mais c'est une bonne habitude à prendre, en effet si on tente d'utiliser une variable qui n'est pas définie, cela plante l'application !



Effectuer les modifications nécessaires sur la vue error.blade.php.

La route /getLogin

```
Route::get('/', function () {  
    return view('home');  
});  
  
// Afficher le formulaire d'authentification  
Route::get('/getLogin', 'UtilisateurController@login');
```

Le rôle de la route /getLogin, sera d'appeler la méthode getLogin() du contrôleur UtilisateurController. On notera qu'il s'agit là aussi d'une méthode GET.



Ajouter la nouvelle route et tester. Normalement on devrait obtenir ceci :

Valider l'authentification

L'utilisateur devra fournir un login et un mot de passe, une lecture sur le login se fera sur la table user et si le mot de passe saisi est identique à celui lue dans la table, alors on placera l'id de l'utilisateur dans la Session et il aura accès aux autres items du menu. Si au contraire il n'y a pas correspondance, un message d'erreur sera affiché sur le formulaire d'authentification.

Pour réaliser la validation de l'authentification, nous aurons besoin des composants suivants :

Composant	Valeur	Rôle
Route	/signIn	Validation de l'authentification
Méthode métier	Login()	Extraction des données Utilisateur et test sur mot de passe
Classes métier	Utilisateur	Classe gérant les méthodes métier
Contrôleur	UtilisateurController	Contrôleur pilotant l'authentification à l'aide de la méthode signIn()

La route /signIn

```
// Réponse au clic sur le bouton Valider du formulaire formLogin  
Route::post('/signIn', 'UtilisateurController@signIn');
```

On notera qu'il s'agit d'une méthode POST, puisque cette route est la réponse au clic sur le bouton valider qui envoie un formulaire.

Remarque : le choix de donner le même nom à la méthode du contrôleur et à la route est purement arbitraire, on aurait pu lui donner un tout autre nom.



Ajouter la nouvelle route.

Les classes métier

Avant d'aborder le détail de la classe Utilisateur, quelques mots sur les choix d'architecture.

Ce qui a présidé ici tout au long de ce TP, c'est le choix de la simplicité tout en conservant bien sûr les principes de l'architecture MVC. Plutôt par exemple d'utiliser la bibliothèque spécialisée de gestion des User fournie avec la plupart des frameworks PHP (et très souvent issue de Symfony), il a été choisi de s'en tenir à une gestion simple de l'authentification. D'une part parce que les besoins sont très simples et surtout d'autre part parce que ces packages (et notamment celui de l'authentification) sont très lourds, très complexes à comprendre et à utiliser et qu'ils n'entrent pas dans l'objectif de ce cours qui est d'illustrer le concept MVC à travers un exemple de framework relativement accessible.

Il y a de même plusieurs façons d'implémenter la partie Model du MVC, l'une assez complexe où l'on va implémenter un diagramme de classes avec toutes les relations (1..*, 1..1, ...) entre les différentes classes. Classes qui avec leurs propriétés (getters et setters) seront essentiellement des DTO (Data Type Object), puis mettre en œuvre une couche DAL (Data Access Layer) qui sera chargée de la persistance, persistance qui sera implémentée par un ORM (ici Eloquent). Si cette architecture se justifie pour des projets conséquents, menés par des développeurs expérimentés, elle est tout à fait inadaptée pour des petits projets réalisés par des développeurs sinon "débutants" du moins relativement peu expérimentés.

L'autre façon d'implémenter la partie Model de l'architecture MVC, c'est de considérer que les classes implémenteront la partie métier, qu'elles gèreront la persistance via un sous-ensemble de Eloquent qu'est QueryBuilder. De cette manière nous manipulerons des objets directement issus de la base de données via un requêteur proche du SQL, gros nous ferons essentiellement du CRUD (Create, Read, Update, Delete). C'est simple, direct et très efficace dans le cadre d'un petit projet.

La classe Utilisateur

Nous allons placer les classes métier dans un dossier que nous nommerons modeles. On y placera ensuite notre première classe que l'on va créer en mode console à l'aide de la commande :

```
php artisan make:model folder/NomClasse
```



Dans le dossier app, créer le dossier modeles, ouvrir une console, descendre dans le dossier mangasworld, saisir la commande : `php artisan make:model modeles/Utilisateur`

```
<?php

namespace App\modeles;

use Illuminate\Database\Eloquent\Model;

class Utilisateur extends Model
{
    //
}
```

La méthode login()

```
<?php
namespace App\modeles;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Session;
use DB;

class Utilisateur extends Model
{
    /**
     * Authentifie l'utilisateur sur son login et Mdp
     * Si c'est OK, son id est enregistré dans la session
     * Cela lui donne accès au menu général (voir page master)
     * @param string $login : Login de l'utilisateur
     * @param string $pwd : MdP de l'utilisateur
     * @return boolean : True or false
     */
    public function login($login, $pwd) {
        $connected = false;
        $user = DB::table('user')
            ->select()
            ->where('login', '=', $login)
            ->first();
        if ($user) {
            // Si le mdp saisi est identique au mdp enregistré
            if ($user->pwd == $pwd) {
                Session::put('id', $user->user_id);
                $connected = true;
            }
        }
        return $connected;
    }
}
```

Cette méthode retourne un booléen qui sera à true si le login et le mot de passe sont corrects et false sinon. Si l'authentification est réussie, on placera l'id de l'utilisateur dans la Session sous la variable de session 'id'. On notera que cela ne peut fonctionner que si le login est unique !

D'autre part comme nous utiliserons le QueryBuilder et les objets de session, nous devrons référencer ces deux packages dans les clauses use.



Compléter la classe Utilisateur.

La méthode signIn() du contrôleur

```
/**
 * Authentifie l'utilisateur
 * @return Vue formLogin ou home
 */
public function signIn() {
    $login = Request::input('login');
    $pwd = Request::input('pwd');
    $utilisateur = new Utilisateur();
    $connected = $utilisateur->login($login, $pwd);
    if ($connected) {
        return view('home');
    } else {
        $erreur = "Login ou mot de passe inconnu !";
        return view('formLogin', compact('erreur'));
    }
}
```

On commence par récupérer les données saisies à l'aide de la méthode `input()` de la classe statique `Request`, puis on instancie un objet `Utilisateur` pour pouvoir appeler sa méthode `login()`. Si cette méthode retourne `true`, c'est que tout est correct, alors on affiche la page d'accueil, sinon on prépare un message d'erreur et on réaffiche le formulaire `formLogin`.

Nous allons devoir référencer la classe `Utilisateur` et modifier la référence au package `Request` car nous utiliserons les méthodes statiques. De plus nous pourrons supprimer la référence au package `Requests` car nous ne retournerons que des vues ou des routes.

```
<?php
namespace App\Http\Controllers;
use Request;
use App\models\Utilisateur;

class UtilisateurController extends Controller
{
```

Remarque : la différence entre la ligne `use Illuminate\Http\Request;` et `use Request;`, c'est que dans le premier cas le framework créera une instance de la classe `Request` et l'injectera dans l'application et dans le deuxième cas il ne fera que référencer la classe `Request` qui devra donc être utilisée comme une classe statique. Cette technologie s'appelle `IoC Inversion of Control`, elle se substitue au développeur pour tout ce qui est instantiation des objets, elle sera étudiée plus tard, pour le moment nous l'utiliserons comme une boîte noire.



Modifier l'entête du contrôleur `UtilisateurController`, saisir la méthode `signIn()` et tester avec le login `auchon` et le mot de passe `xxx`, puis avec le bon mot de passe (`paul`). Dans premier cas le message d'erreur devrait s'afficher et dans le deuxième la copie d'écran ci-dessous :

Bienvenue sur Mangas World !

La déconnexion

Elle consiste à supprimer la variable de session id, ce qui aura pour conséquence de supprimer l'accès aux items de menu et permettre à nouveau la connexion.

La route signOut

```
// Déloguer l'utilisateur
Route::get('/signOut', 'UtilisateurController@signOut');
```

La méthode signOut() du contrôleur

```
/**
 * Déconnecte le visiteur authentifié
 * @return Vue home
 */
public function signOut() {
    $utilisateur = new Utilisateur();
    $utilisateur->logout();
    return view('home');
}
```

La méthode logout() de la classe Utilisateur

```
/**
 * Délogue l'utilisateur en supprimant son Id
 * de la session => le menu n'est plus accessible
 */
public function logout() {
    Session::forget('id');
}
```

Remarque : on aurait pu directement mettre le `Session::forget('id');` dans le contrôleur, mais cela n'aurait pas été une bonne pratique. En effet on peut considérer que ce n'est pas de la responsabilité du contrôleur de nettoyer l'environnement propre à l'utilisateur mais à la classe Utilisateur elle-même. De plus pour le moment cela se réduit à une seule ligne, mais rien ne nous dit qu'il n'y aura pas à l'avenir d'autres choses à faire comme par exemple vider un panier, fermer des connexions ...



Effectuer les ajouts demandés et tester une déconnexion.

Liste de tous les mangas

Pour réaliser la liste de tous les Mangas (item Lister du menu) nous aurons besoin des composants suivants :

Composant	Valeur	Rôle
Route	/listMangas	Prise en compte du clic sur l'item Liste du menu
Vue	listeMangas.blade.php	Affichage de la liste des mangas
Méthode métier	getMangas()	Extraction des mangas et données issues des tables liées
Classes métier	Manga, Dessinateur, Scenariste, Genre	Conteneurs des méthodes métier
Contrôleur	MangaController	Pilotage de la demande de liste à l'aide de la méthode getMangas()

La vue listeMangas

Le cœur de cette vue est constitué par la boucle foreach qui va parcourir la collection \$mangas et injecter pour chacun des items les données dans le composant table :

```
@foreach($mangas as $manga)
<tr>
    <td> {{ $manga->id_manga }} </td>
    <td> {{ $manga->titre }} </td>
    <td> {{ $manga->lib_genre }} </td>
    <td> {{ $manga->nom_dessinateur }} </td>
    <td> {{ $manga->nom_scenariste }} </td>
    <td> {{ $manga->prix }} </td>
    <td style="text-align:center;"><a href="{{ url('/modifierManga') }}/{{ $manga->id_manga }}">
        <span class="glyphicon glyphicon-pencil" data-toggle="tooltip" data-placement="top" title="Modifier">
    <td style="text-align:center;">
        <a class="glyphicon glyphicon-remove" data-toggle="tooltip" data-placement="top" title="Supprimer"
            onclick="javascript:if (confirm('Suppression confirmée ?')) { window.location='{{ url('/supprimerManga') }}/{{ $manga->id_manga }}';}">
        </a>
    </td>
</tr>
@endforeach
```

Important : bien que les noms utilisés y incitent (\$mangas et \$mangas), il ne faut pas considérer qu'il s'agit d'une collection d'objets de type Manga. En effet d'une part la classe Manga ne possède pas la propriété nom_dessinateur par exemple et d'autre part cette collection est le résultat d'une requête combinant via des jointures plusieurs tables (nous verrons cela plus loin).

Cette liste donnera accès à la modification et à la suppression d'un manga.

Modification

```
<td style="text-align:center;"><a href="{{ url('/modifierManga') }}/{{ $manga->id_manga }}">
    <span class="glyphicon glyphicon-pencil" data-toggle="tooltip" data-placement="top" title="Modifier">
</td>
```

On appellera la Route /modifierManga en lui passant en paramètre l'id du manga que l'on veut modifier.

Suppression

```
<td style="text-align:center;">
    <a class="glyphicon glyphicon-remove" data-toggle="tooltip" data-placement="top" title="Supprimer" href="#"
        onclick="javascript:if (confirm('Suppression confirmée ?'))
            { window.location='{{ url('/supprimerManga') }}/{{ $manga->id_manga }}';}">
    </a>
</td>
```

Là, c'est un peu plus compliqué, comme une suppression est définitive, on va demander confirmation à l'utilisateur via une méthode javascript (confirm('message')), si la réponse est true, alors on va avec window.location lancer l'url /supprimerManga avec en paramètre l'id du manga à supprimer, si la réponse est false, on ne fait rien.



Récupérer la vue listeMangas.blade.php dans le dossier Ressources, la copier dans le dossier views et la compléter en utilisant les informations ci-dessus.

Les classes métier

Chaque table sera représentée par une classe portant son nom, et comme pour la classe Utilisateur, elles implémenteront les méthodes métier et la gestion de la persistance.

La classe Manga

```
<?php
namespace App\modeles;
use Illuminate\Database\Eloquent\Model;
use DB;

class Manga extends Model
{
}
```

La classe Dessinateur

```
<?php
namespace App\modeles;
use Illuminate\Database\Eloquent\Model;
use DB;

class Dessinateur extends Model
{
}
```

La classe Scenariste

```
<?php
namespace App\modeles;
use Illuminate\Database\Eloquent\Model;
use DB;

class Scenariste extends Model
{
}
```

La classe Genre

```
<?php
namespace App\modeles;
use Illuminate\Database\Eloquent\Model;
use DB;

class Genre extends Model
{
}
```



Créer les classes Manga, Dessinateur, Genre et Scenariste en utilisant les informations fournies ci-dessus.

La méthode métier getMangas()

```
/**
 * Lecture de tous les mangas avec mise en oeuvre
 * des jointures
 * @return Collection de Manga
 */
public function getMangas() {
    $mangas = DB::table('manga')
        ->Select('id_manga', 'titre', 'genre.lib_genre', 'dessinateur.nom_dessinateur',
            'scenariste.nom_scenariste', 'prix')
        ->join('genre', 'manga.id_genre', '=', 'genre.id_genre')
        ->join('dessinateur', 'manga.id_dessinateur', '=', 'dessinateur.id_dessinateur')
        ->join('scenariste', 'manga.id_scenariste', '=', 'scenariste.id_scenariste')
        ->get();
    return $mangas;
}
```

On notera que l'on marque que l'on retourne une collection en utilisant le pluriel (\$mangas), ce n'est pas une obligation, mais c'est une bonne pratique.



Saisir la méthode getMangas() dans la classe Manga.

Le contrôleur MangaController

```
/**
 * Affiche la liste de tous les Mangas
 * Si la Session contient un message d'erreur,
 * on le récupère et on le supprime de la Session
 * @return Vue listerMangas
 */
public function getMangas() {
    $erreur = Session::get('erreur');
    Session::forget('erreur');
    $manga = new Manga();
    // On récupère la liste de tous les mangas
    $mangas = $manga->getMangas();
    // On affiche la liste de ces mangas
    return view('listeMangas', compact('mangas', 'erreur'));
}
```

On constate que l'on appelle la méthode `getMangas()` de la classe `Manga` après que l'on ait instancié un objet de cette classe. Ensuite on appelle la vue `listeMangas` en lui passant la collection `$mangas` retournée par la méthode `getMangas()`.

On constatera que l'on passe via la fonction `compact()` un tableau associatif constitué de deux occurrences à la vue `listeMangas`. La première occurrence est constituée du tableau d'objets retourné par la méthode `getmangas()`.

Pour le moment nous laisserons de côté la gestion des erreurs, nous aborderons ce sujet plus tard.



Créer le contrôleur `MangaController` à l'aide de la commande `Laravel`, puis saisir le code ci-dessus.

Attention : puisqu'on utilise les objets de session, il faudra faire référence à la bibliothèque de gestion des sessions comme on l'a fait dans le Modèle Utilisateur.













La route `/listeMangas`

```
// Afficher la liste de tous les Mangas
Route::get('/listerMangas', 'MangaController@getMangas');
```



Rajouter la route de type `get` qui appelle la méthode `getMangas` du contrôleur `MangaController`, puis tester. Normalement le navigateur devrait afficher quelque chose proche de la copie ci-dessous.

Liste de mes Mangas

Id	Titre	Genre	Dessinateur	Scénariste	Prix	Modification	Suppression
1	Akatsuki Vol.2	Aventure	TITE	TITE	12.50		
2	Collège Fou Fou Fou (le)	Tanche-de-vie	ONE	ONE	10.90		
3	Yu-Gi-Oh ! 5D's Vol.9	Aventure	TORIYAMA	YUSUKE	8.75		
4	Hack - Le bracelet du crépuscule	Aventure	OBA	IWAAKI	9.90		
5	7 Yakuzas	Action	OBATA	TOGASHI	12.25		
6	7 milliards d'aiguilles	Policier	TORIYAMA	TOGASHI	11.78		

Liste des Mangas d'un genre

Nous ne voulons plus lister tous les Mangas, mais seulement les Mangas d'un genre que l'on aurait choisi dans une liste. Il nous faut d'abord afficher la liste des genres pour pouvoir en choisir un. Pour réaliser cette fonctionnalité nous avons besoin des composants suivants :

Composant	Valeur	Rôle
Route	/listerGenres	Prise en compte du clic sur l'item Liste par genre du menu
Formulaire	formGenre.blade.php	Affichage de la liste des genres et sélection d'un genre
Méthode métier	getGenres()	Extraction des genres de la table genre
Classes métier	Genre	Conteneur des méthodes métier
Contrôleur	GenreController	Pilotage de la demande de formulaire à l'aide de la méthode getGenres().

Le formulaire formGenre

```

{!! Form::open(['url' => 'listMangasGenre']) !!}
<div class="col-md-12 well well-sm">
    <center><h1>Choix d'un Genre</h1></center>
    <div class="form-horizontal">
        <div class="form-group">
            <label class="col-md-3 control-label">Genre : </label>
            <div class="col-md-3">
                <select class="form-control" name="cbGenre" required>
                    <OPTION VALUE=0>Sélectionner un genre</option>
                    @foreach ($genres as $genre)
                        <option value="{{ $genre->id_genre }}"> {{ $genre->lib_genre}} </option>
                    @endforeach
                </select>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-6 col-md-offset-3">
                <button type="submit" class="btn btn-default btn-primary"><span class="glyphicon glyphicon-send">
                </button>
                <button type="button" class="btn btn-default btn-primary">
                    onclick="javascript: window.location = '{{ url('/') }}';">
                    <span class="glyphicon glyphicon-remove"></span>
                    Annuler
                </button>
            </div>
        </div>
    </div>

```

Il s'agit d'un formulaire permettant la sélection du genre, l'envoi du formulaire (bouton submit) sera traité par la route /listMangasGenre qui elle-même fera appel au contrôleur MangaController.

Il n'y a pas de difficulté particulière, la liste déroulante est construite par un foreach qui puise les données dans la collection \$genres qui là est une collection d'objets de type Genre.

Nous verrons plus loin l'explication du javascript associé au bouton Annuler.



Récupérer la vue formGenre du dossier Ressources et la compléter.

La méthode métier getGenres()

```

/**
 * Liste des genres
 * @return collection de Genre
 */
public function getGenres() {
    $genres = DB::table('genre')->get();
    return $genres;
}

```

Elle est simpliste et se trouve dans la classe Genre. A noter que l'on aurait très bien pu écrire :
return DB::table('genre')->get(); mais cela aurait été moins pratique en cas de débogage !

C'est d'ailleurs une leçon à retenir : les écritures synthétiques de code ne facilitent pas le débogage car elles ne permettent pas de placer des points d'arrêt pour visualiser les valeurs obtenues. Et d'autre part elles sont moins facilement lisibles donc moins facilement maintenables et il FAUT TOUJOURS penser à la MAINTENANCE du code que l'on écrit !



Ajouter la méthode métier à la classe Genre.

Le contrôleur GenreController

Avant de commencer demandons-nous pourquoi on n'a pas mis la production de la vue formGenre dans le contrôleur MangaController puisqu'au final c'est ce contrôleur qui sera invoqué pour produire la liste des Mangas d'un genre !

Parce que tout simplement cela ne relève pas de la responsabilité du contrôleur associé à la classe Manga de s'occuper de gérer les fonctionnalités d'une autre classe. Donc c'est bien au contrôleur GenreController de produire la liste des genres et ce sera au contrôleur MangaController de produire la liste des mangas d'un genre. A chacun ses responsabilités.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Support\Facades\Session;
use App\models\Genre;

class GenreController extends Controller
{
    /**
     * Afficher les genres dans une liste déroulante
     * @return Vue formGenre
     */
    public function getGenres() {
        $erreur = Session::get('erreur');
        Session::forget('erreur');
        $genre = new Genre();
        $genres = $genre->getGenres();
        return view('formGenre', compact('genres', 'erreur'));
    }
}
```



Créer le contrôleur à l'aide de la commande Laravel, puis saisir le code ci-dessus.

La route listerGenres

```
// Afficher la liste déroulante des genres
Route::get('/listerGenres', 'GenreController@getGenres');
```



Ajouter la route et tester.

[Mangas World](#) [Lister](#) [Mangas par genre](#) [Ajouter](#)

Choix d'un Genre

Genre :

Liste des Mangas du genre sélectionné

Pour réaliser cette fonctionnalité nous avons besoin des composants suivants :

Composant	Valeur	Rôle
Route	/listerMangasGenre	Prise en compte de l'appel du contrôleur MangaGenre
Vue	listeMangas.blade.php	La même que celle utilisée pour afficher tous les Mangas
Méthode métier	getMangasGenre()	Extraction des mangas ayant le genre choisi
Classes métier	Manga	Conteneur des méthodes métier
Contrôleur	MangaController	Pilotage de la demande de liste à l'aide de la méthode getMangasGenre()

La méthode métier getMangasGenre()

```
/**
 * Lecture de tous les mangas d'un genre
 * @param int $id_genre : id du genre
 * @return Collection de Manga
 */
public function getMangasGenre($id_genre) {
    $mangas = DB::table('manga')
        ->select('id_manga', 'titre', 'genre.lib_genre', 'dessinateur.nom_dessinateur',
            'scenariste.nom_scenariste', 'prix')
        ->where('manga.id_genre', '=', $id_genre)
        ->join('genre', 'manga.id_genre', '=', 'genre.id_genre')
        ->join('dessinateur', 'manga.id_dessinateur', '=', 'dessinateur.id_dessinateur')
        ->join('scenariste', 'manga.id_scenariste', '=', 'scenariste.id_scenariste')
        ->get();
    return $mangas;
}
```

On remarquera la position un peu inhabituelle du where qui en fait est tout à fait compatible avec le format SQL92 du langage SQL.



Saisir la méthode `getMangasGenre()` dans la classe `Manga`.

Le contrôleur `MangaController`

Nous allons lui ajouter la méthode `getMangasGenre()`.

```
/**
 * Afficher la liste des tous les Mangas d'un Genre
 * Si on a sélectionné un genre, on récupère tous les
 * mangas de ce genre et on les affiche
 * Si on n'a pas sélectionné de genre, on place
 * un message d'erreur dans la Session et on
 * relance le formulaire de sélection d'un genre
 * @return Vue listerMangas
 */
public function getMangasGenre() {
    $erreur = "";
    // On récupère l'id du genre sélectionné
    $id_genre = Request::input('cbGenre');
    // Si on a un id de genre
    if ($id_genre) {
        $manga = new Manga();
        // On récupère la liste de tous les mangas du genre choisi
        $mangas = $manga->getMangasGenre($id_genre);
        // On affiche la liste de ces mangas
        return view('listeMangas', compact('mangas', 'erreur'));
    } else {
        $erreur = "Il faut sélectionner un genre !";
        Session::put('erreur', $erreur);
        return redirect('/listerGenres');
    }
}
```

On commence par récupérer l'id du genre sélectionné, puis on appelle la méthode `getMangasGenre()` qui retourne la collection de `Mangas` en lui passant en paramètre d'id du genre choisi. Enfin on appelle la vue `listeMangas` en lui passant la collection `mesMangas`.

Messages d'erreur

Test de validité côté serveur

Pourquoi a-t-on besoin de tester que l'utilisateur a bien saisi un item dans la liste des genres ? Il suffirait d'ajouter l'attribut `required` dans la vue comme pour les input type text ! Le problème c'est que cet attribut vérifie que l'on ait rien sélectionné, or la liste est pré-positionnée sur l'item 0, la preuve :

```
<select class='form-control' name='cbGenre' required>
    <OPTION VALUE=0>Sélectionner un genre</option>
    @foreach ($genres as $genre)
        <option value="{{ $genre->id_genre }}"> {{ $genre->
    @endforeach
</select>
```

TD Mangas World

Du coup on peut mettre tous les required que l'on veut cela ne servira à rien, il faut donc effectuer le test de validité côté serveur.

Gérer les messages d'erreur

Comment passer des données lorsqu'on invoque une redirection vers une route, ce qui est le cas si aucun genre n'a été sélectionné (`return redirect('/listGenres');`) ?

Le plus simple est de placer ces données dans un espace commun à l'ensemble des composants et cet espace commun c'est la Session. On va donc placer le message d'erreur dans une variable de Session et effectuer la redirection, à charge pour la méthode qui sera appelée de récupérer ce message. C'est ce qui est fait dans :

```
public function getGenres() {  
    $erreur = Session::get('erreur');  
    Session::forget('erreur');  
    $genre = new Genre();  
}
```

Le problème avec la Session, c'est qu'elle peut très vite se remplir et donc obérer les performances du serveur, il faut donc libérer cette ressource dès que l'on n'en a plus besoin, c'est ce qui est fait avec la méthode `forget('erreur')`.



Ajouter la méthode au contrôleur MangaController.

La route /listMangasGenre

Lorsqu'on va cliquer sur le bouton valider du formulaire `formGenre`, c'est cette route qui est invoquée.

```
// Lister tous les mangas d'un genre sélectionné  
Route::post('/listMangasGenre', 'MangaController@getMangasGenre');
```



Ajouter la route et tester en sélectionnant un genre puis sans sélectionner de genre pour voir le message d'erreur.

Mangas World							
Lister Mangas par genre Ajouter							
Liste de mes Mangas							
Id	Titre	Genre	Dessinateur	Scénariste	Prix	Modification	Suppression
1	Akatsuki Vol.2	Aventure	TITE	TITE	12.50		
3	Yu-Gi-Oh ! 5D's Vol.9	Aventure	TORIYAMA	YUSUKE	8.75		
4	Hack - Le bracelet du crépuscule	Aventure	OBA	IWAAKI	9.90		

Modifier un Manga

Nous allons voir maintenant comment modifier un Manga qui aura été sélectionné dans l'une des deux listes de Mangas.

Afficher le Manga sélectionné.

La première étape d'une modification consiste à afficher le Manga sélectionné en plaçant les données dans un formulaire. Pour cela nous aurons besoin des composants suivants :

Composant	Valeur	Rôle
Route	/modifierManga/{id}	Prise en compte du clic sur l'item sélectionné dans la liste des Mangas
Formulaire	formManga	Affichage le manga et permet la saisie
Méthodes métier	getManga()	Extraction du manga sélectionné
	getGenres()	Extraction de la liste des genres
	getDessinateurs()	Extraction de la liste des dessinateurs
	getScenaristes()	Extraction de la liste des scénaristes
Classes métier	Manga, Dessinateur, Scenariste, Genre	Conteneur des méthodes métier
Contrôleur	MangaController	Pilotage de la demande de construction du formulaire à l'aide de la méthode updateMangas()

Le formulaire formManga

le formulaire est trop important pour être présenté in extenso, nous allons donc étudier quelques parties significatives, vous en déduirait le reste.

L'envoi du formulaire

```
@section('content')
{!! Form::open(['url' => 'validerManga', 'files' => true]) !!}
<div class="col-md-12 well well-sm">
```

Lorsqu'on cliquera sur le bouton Valider ce sera la route validerManga qui sera invoquée. Pour le moment nous laisserons de côté le paramètre files.

L'id du manga à modifier

Lorsqu'on voudra faire la requête de mise à jour, nous aurons besoin de l'id du manga pour faire un update ... where id_manga =. Il y a plusieurs façons de procéder l'une d'entre elle consiste à placer la valeur de l'id dans un champ caché (input type hidden), puis de le récupérer au moment de la validation, une autre consisterait à passer l'id en paramètre à la route validerManga. ici nous prendrons la première solution :

```
<div class="form-group">
    <input type="hidden" name="id_manga" value="{{ $manga->id_manga or 0 }}" />
```

A noter que l'on peut placer les input hidden n'importe où car ils sont par définition non visibles !

Dépôt d'une valeur dans un input

```
<div class="col-md-3">
    <input type="text" name="titre"
        value="{{ $manga->titre or '' }}" class="form-control" required autofocus>
</div>
```

TD Mangas World

La syntaxe est classique, mais ici on lui a apporté une variante, le or " qui signifie que si la propriété ou la variable invoquée (ici \$manga->titre) n'est pas définie, il faudra mettre la valeur chaîne vide (on aurait pu mettre autre chose). Cela nous servira essentiellement lors de l'ajout d'un Manga, car nous réutiliserons le même formulaire et sans cette fonctionnalité de substitution, l'application pourrait "planter".

Le cas des listes déroulantes

```
<label class="col-md-3 control-label">Genre : </label>
<div class="col-md-3">
    <select class='form-control' name='cbGenre' required>
        <OPTION VALUE=0>Sélectionner un genre</option>
        @foreach ($genres as $genre)
            selected=""
            <option value="{{ $genre->id_genre }}"
                @if( $genre->id_genre == $manga->id_genre )
                    selected="selected"
                @endif
            > {{ $genre->lib_genre}} </option>
        @endforeach
    </select>
</div>
```

Lorsqu'on fait une modification, il faut pouvoir positionner la liste déroulante sur le bon item, c'est la raison d'être du @if. Les deux autres listes (Dessinateur et Scénariste) seront construites sur le même modèle.

L'image de la couverture

```
<label class="col-md-3 control-label">Couverture : </label>
<div class="col-md-6">
    <input type="hidden" name="MAX_FILE_SIZE" value="204800"/>
    <input name="couverture" type="file" class="btn btn-default pull-left"/>
    <input type="hidden" name="couvertureHidden" value="{{ $manga->couverture or '' }}" />
    couverture }}" />
</div>
```

Pour le moment nous nous contenterons de dire que l'on place le nom de l'image dans un INPUT HIDDEN pour pouvoir le récupérer en temps utile, nous verrons plus tard comment gérer l'upload d'une image.

Validation et annulation

```
<button type="submit" class="btn btn-default btn-primary">
    <span class="glyphicon glyphicon-ok"></span> Valider
</button>
&nbsp;
<button type="button" class="btn btn-default btn-primary"
    onclick="javascript: window.location = '{{ url('/') }}';">
    <span class="glyphicon glyphicon-remove"></span> Annuler
</button>
```

TD Mangas World

La validation ne pose pas de problème, c'est un classique INPUT type submit. Pour l'annulation, le choix a été fait de faire appel à l'objet javascript location qui effectue une redirection vers l'url qu'on lui affecte via la méthode assign() qui est invoquée par défaut. Pour plus de détail voir : <https://developer.mozilla.org/fr/docs/Web/API/window/location>.



Récupérer le formulaire formManga.blade.php dans les Ressources et le compléter,
Copier le dossier images depuis les Ressources et le coller dans le dossier public.

Les méthodes métier

Les méthodes métier pour produire les listes déroulantes sont toutes basées sur le même principe. Nous en avons vu une : getGenres(), il suffira de s'en inspirer.



Créer les méthodes métier getDessinateurs() et getScenaristes().

Le contrôleur MangaController

Il faudra ajouter la méthode addManga() qui va lire le Manga à afficher et appeler toutes les méthodes métier qui alimenteront les listes déroulantes.

```
/**
 * Initialise toutes les listes déroulantes
 * Lit le manga à modifier
 * Initialise le formulaire en mode Modification
 * @param int $id Id du Manga à modifier
 * @param string $erreur message d'erreur (paramètre optionnel)
 * @return Vue formManga
 */
public function updateManga($id, $erreur = "") {
    $leManga = new Manga();
    $manga = $leManga->getManga($id);
    $genre = new Genre();
    $genres = $genre->getGenres();
    $dessinateur = new Dessinateur();
    $dessinateurs = $dessinateur->getDessinateurs();
    $scenariste = new Scenariste();
    $scenaristes = $scenariste->getScenaristes();
    $titreVue = "Modification d'un Manga";
    // Affiche le formulaire en lui fournissant les données à afficher
    return view('formManga', compact('manga', 'genres', 'dessinateurs',
        'scenaristes', 'titreVue', 'erreur'));
}
```

Nous verrons plus tard la raison de la présence du deuxième paramètre et pourquoi il est initialisé.



Saisir la méthode updateManga().

La méthode getManga(\$idManga)

```
/**
 * Lecture d'un manga sur son Id
 * @param int $idManga à lire
 * @return Objet Manga
 */
public function getManga($idManga) {
    $manga = DB::table('manga')
        ->select()
        ->where('id_manga', '=', $idManga)
        ->first();
    return $manga;
}
```



Saisir la méthode getManga() dans la classe Manga.

La route /modifierManga/{id}

```
// Afficher un manga pour pouvoir le modifier
Route::get('/modifierManga/{id}', 'MangaController@updateManga');
```

Il s'agit de notre première route avec passage de paramètre, elle répond au clic sur le bouton Modifier structuré comme suit dans la vue listerMangas :

```
<a href="{{ url('/modifierManga') }}/{{ $manga->id_manga }}">
```

Ce qui donnera par exemple dans le navigateur :

<http://localhost/arsane/mangasworld/public/modifierManga/4>



Saisir la route et tester l'affichage d'un manga sélectionné dans une des listes.

Modification d'un Manga

Titre :

Genre :

Tanche-de-vie
▼

Scenariste :

ONE
▼

Dessinateur :

ONE
▼

Prix :

Couverture :

Parcourir...

Aucun fichier sélectionné.



✓ Valider

✕ Annuler

Valider les modifications

Lorsqu'on va cliquer sur le bouton Valider, il faudra récupérer les données, effectuer la mise à jour dans la base de données et réafficher la liste des Mangas.

Pour cela nous aurons besoin des composants suivants :

Composant	Valeur	Rôle
Route	/validerManga	Prise en compte du clic sur le bouton Valider du formulaire
Méthode métier	updateManga()	Mise à jour des données dans la base de données
Contrôleur	MangaController	Méthode validateManga() qui récupère les données saisies et appelle la mise à jour

La route

```
// Enregistrer la mise à jour d'un manga
Route::post('/validerManga', 'MangaController@validateManga');
```



Ajouter la route.

Méthode updateManga() de la classe Manga

```
/**
 * Mise à jour d'un Manga sur son Id
 * @param int $id_manga
 * @param string $titre
 * @param string $couverture
 * @param decimal $prix
 * @param int $id_dessinateur
 * @param int $id_genre
 * @param int $id_scenariste
 */
public function updateManga($id_manga, $titre, $couverture, $prix, $id_dessinateur, $id_genre, $id_scenariste) {
    try {
        DB::table('manga')->where('id_manga', '=', $id_manga)
            ->update(['id_dessinateur' => $id_dessinateur, 'prix' => $prix,
                    'titre' => $titre, 'couverture' => $couverture,
                    'id_genre' => $id_genre, 'id_scenariste' => $id_scenariste]);
    } catch (Exception $ex) {
        throw $ex;
    }
}
```

La nouveauté ici, c'est faisant une opération sur la base de données susceptible de provoquer des erreurs fatales (contraintes de colonnes, contraintes d'intégrité référentielle ...), on est tenu de la placer dans une gestion des exceptions. Ici on se contentera de propager l'exception vers le niveau supérieur.

Important : pour pouvoir gérer les exceptions, il faut ajouter la clause `use Exception;` à la classe.



Saisir la méthode métier `updateManga()`.

Méthode validateManga() du contrôleur

```
/**
 * Enregistre une mise à jour d'un Manga
 * Si la modification d'un Manga
 * provoque une erreur fatale, on la place
 * dans la Session et on réaffiche le formulaire
 * Sinon réaffiche la liste des mangas
 * @return Redirection listerMangas
 */
public function validateManga() {
    // Récupération des valeurs saisies
    $id_manga = Request::input('id_manga'); // id dans le champs caché
    $id_dessinateur = Request::input('cbDessinateur'); // Liste déroulante
    $prix = Request::input('prix');
    $id_scenariste = Request::input('cbScenariste'); // Liste déroulante
    $titre = Request::input('titre');
    $id_genre = Request::input('cbGenre'); // Liste déroulante
    // Si on a uploadé une image, il faut la sauvegarder
    // Sinon on récupère le nom dans le champ caché
    if (Request::hasFile('couverture')) {
        $image = Request::file('couverture');
        $couverture = $image->getClientOriginalName();
        Request::file('couverture')->move(base_path() . '/public/images/', $couverture);
    } else {
        $couverture = Request::input('couvertureHidden');
    }
    $manga = new Manga();
    try {
        $manga->updateManga($id_manga, $titre, $couverture, $prix, $id_dessinateur, $id_genre, $id_scenariste);
    } catch (Exception $ex) {
        $erreur = $ex->getMessage();
        return $this->updateManga($id_manga, $erreur);
    }
    // On réaffiche la liste des mangas
    return redirect('/listerMangas');
}
```

Là aussi pas de difficulté particulière. On récupère l'Id du manga dans le champ HIDDEN, idem pour le nom de l'image de la couverture, on appelle la mise à jour puis on redirige sur la liste des Mangas.

La gestion des images

On commence par tester si l'input de type file possède bien un fichier qui a été sélectionné par l'internaute. Pour ce faire il nous a fallu placer l'argument file dans le formulaire, indiquant que l'on souhaite gérer les fichiers uploadés :

```
@section('content')
{!! Form::open(['url' => 'validerManga', 'files' => true]) !!}
```

S'il y a bien un fichier, on récupère l'objet, c'est d'ailleurs là que le fichier est réellement téléchargé en mémoire. On récupère son nom (celui qu'il avait sur le disque dur de l'internaute) et enfin à l'aide de la méthode move() on l'enregistre dans le dossier images.

La gestion de l'image nécessite de prendre en compte soit qu'elle n'existait pas et que l'utilisateur vient de l'uploader il faut alors finaliser son téléchargement, soit qu'elle existait il faut alors penser récupérer son nom qui a été placé dans le champ caché car dans tous les cas le nom de cette image sera enregistré dans la base de données.

L'enregistrement

Comme on appelle une méthode (`updateManga()`) susceptible de propager une exception, il nous faut encadrer cet appel dans un gestionnaire d'exceptions. S'il y a erreur fatale notre traitement de l'exception se limitera à récupérer le message d'erreur originel et à la réafficher dans le formulaire. Evidemment il ne faudrait pas faire cela sur une application en production, mais cela nous suffira en ce nous concerne. On rebouclera ensuite sur la modification en réappelant la méthode `updateManga()`.

Les paramètres optionnels

On remarquera que l'on passe la variable `$erreur` en paramètre, ce qui nous amène à expliquer la raison d'être du deuxième paramètre de la méthode `updateManga($id, $erreur = "")`.

Le PHP admet les paramètres optionnels sous réserve que ceux-ci soient initialisés dans la signature de la fonction. Donc si l'appel se fait avec seulement le premier paramètre, ce qui était le cas lors du premier appel, alors la variable `$erreur` est initialisée à vide et donc définie, si au contraire la méthode est appelée avec deux paramètres alors l'initialisation est ignorée et la variable `$erreur` conservera la valeur qui lui a été fournie lors de l'appel, dans le cas présent le message d'erreur. Dans tous les cas la variable `$erreur` pourra être utilisée à l'intérieur de la méthode appelée.

Important : il faudra là aussi ajouter la clause `use Exception;` au contrôleur.



Saisir la méthode `validateManga()` et tester une mise à jour normale, puis avec un prix à zéro.

Ajout d'un Manga

Pour effectuer un ajout nous aurons besoin des composants suivants :

Composant	Valeur	Rôle
Formulaire	<code>formManga</code>	C'est le même que pour la modification
Routes	<code>/ajouterManga</code>	Prise en compte du clic sur l'item Ajouter du menu
	<code>/validerManga</code>	C'est la même que tout la mise à jour
Méthode métier	<code>addManga()</code>	Insertion des données dans la base de données
Contrôleur	<code>MangaController</code>	Méthode <code>addManga()</code> qui initialisera un Manga vide ainsi que les listes déroulantes Modification de la méthode <code>validateManga()</code> pour prendre en compte l'insertion

Comme pour la mise à jour, l'ajout se fera en deux étapes, d'abord afficher le formulaire avec les champs vides sauf pour les listes déroulantes, saisie des données et insertion après validation.

Méthode addManga()

```
/**
 * Initialise toutes les listes déroulante
 * Place le formulaire formManga en mode ajout
 * @param string $erreur message d'erreur (paramètre optionnel)
 * @return Vue formManga
 */
public function addManga($erreur = "") {
    $manga = new Manga();
    $genre = new Genre();
    $genres = $genre->getGenres();
    $dessinateur = new Dessinateur();
    $dessinateurs = $dessinateur->getDessinateurs();
    $scenariste = new Scenariste();
    $scenaristes = $scenariste->getScenaristes();
    $titreVue = "Ajout d'un Manga";
    // Affiche le formulaire en lui fournissant les données à afficher
    return view('formManga', compact('manga', 'genres', 'dessinateurs', 'scenaristes', 'titreVue', 'erreur'));
}
```

Il n'y a pas de difficulté particulière, nous avons déjà vu cela dans la demande de modification. La seule chose qui mérite explication, c'est la raison pour laquelle on fait un `new Manga()`, normalement on ne devrait pas en avoir besoin. C'est essentiellement pour initialiser les champs du formulaire même si en réalité dès lors qu'on a mis dans les INPUT du formulaire le `or ''` ou `or 0`, il n'est pas vraiment nécessaire de passer l'objet `$manga`, mais là aussi c'est une bonne pratique de ne pas faire une confiance aveugle à ce qui "devrait être", mais plutôt de systématiquement fiabiliser et sécuriser les composants qu'on développe.

Donc le champ HIDDEN `id_manga` contiendra la valeur explicite de 0 quand on demandera une création et une autre valeur lorsqu'il s'agira d'une modification.

Or comme nous allons utiliser la même méthode (`validateManga()`) pour enregistrer un ajout, nous nous servirons de cette valeur pour différencier l'enregistrement d'un ajout de celui d'une modification.



Saisir la méthode `addManga()`

La route `ajouterManga`

Cette route appelle la méthode `addManga` du contrôleur `MangaController`.



Ajouter la route `/ajouterManga`.

La méthode métier

```
/**
 * Insertion d'un manga
 * Note : la clé primaire (id_manga) est en auto-incrément
 * @param string $titre
 * @param string $couverture
 * @param decimal $prix
 * @param int $id_dessinateur
 * @param int $id_genre
 * @param int $id_scenariste
 */
public function insertManga($titre, $couverture, $prix, $id_dessinateur, $id_genre, $id_scenariste) {
    try {
        DB::table('manga')->insert([
            'id_dessinateur' => $id_dessinateur, 'prix' => $prix,
            'titre' => $titre, 'couverture' => $couverture,
            'id_genre' => $id_genre, 'id_scenariste' => $id_scenariste
        ]);
    } catch (Exception $ex) {
        throw $ex;
    }
}
```


Il n'y a pas de difficulté particulière si ce n'est que l'on ne spécifie pas la clé primaire car la colonne `id_manga` de la table `manga` est en auto-incrément.

 Saisir la méthode métier `insertManga()`.

La méthode `validateManga()`


```
// Si l'Id est > 0 c'est une mise à jour
// sinon c'est une insertion
try {
    if ($id_manga > 0) {
        $manga->updateManga($id_manga, $titre, $couverture, $prix, $id_dessinateur, $id_genre, $id_scenariste);
    } else {
        $manga->insertManga($titre, $couverture, $prix, $id_dessinateur, $id_genre, $id_scenariste);
    }
} catch (Exception $ex) {
    $erreur = $ex->getMessage();
    if ($id_manga > 0) {
        return $this->updateManga($id_manga, $erreur);
    } else {
        return $this->addManga($erreur);
    }
}
```

Comme cela a été spécifié plus haut on va se servir de la valeur de `id_manga` pour faire la différence entre une insertion et une mise à jour. Une valeur 0 signifie qu'il s'agit d'une insertion et une valeur supérieure à 0 qu'il s'agit d'une mise à jour.

 Modifier la méthode `validateManga()` et tester un ajout, tester aussi un ajout sans choisir par exemple un Dessinateur pour vérifier que la gestion des exceptions fonctionne bien.

Supprimer un Manga

Avec tout ce que nous avons fait jusqu'à présent, vous disposez de l'ensemble des connaissances pour réaliser par vous-même cette fonctionnalité !

 Au travail ;-)

Un peu de sécurité

Nous avons limité l'accès aux fonctionnalités proposées par cette application en n'affichant pas les items de menus si l'utilisateur n'était pas authentifié, mais est-ce suffisant ?

Pour le savoir, faisons une petite expérience. Nous allons nous connecter, effectuer une requête sur la liste des mangas et ce faisant copier l'url menant à cette liste, nous déconnecter et tenter d'utiliser cette url (<http://localhost/XXXXX/mangasworld/public/listerMangas>) pour accéder à la liste des Mangas sans s'être authentifié.



Effectuer les opérations demandées.

On constate qu'on a bien accès à la liste des Mangas bien qu'on est pas été autentifé !

Il nous faut donc résoudre ce problème de sécurité et pour ce faire, nous allons utiliser un MiddleWare.

Les MiddleWare

Comme leur nom l'indique un MiddleWare est un composant qui va se situer au milieu et en l'occurrence entre l'arrivée d'une requête HTTP et sa prise en compte par le routeur. Son rôle est d'intercepter l'objet REQUEST et d'effectuer des actions la plupart du temps avant que l'url ne soit traitée, mais parfois aussi après.

Création

Nous allons donc créer le MiddleWare Autorisation qui va tester si l'id de l'utilisateur est bien dans l'objet Session, ce qui (pour nous) sera suffisant pour indiquer s'il s'est connecté ou non.

```
php artisan make:middleware NomMiddleWare
```

Les MiddleWare se créent systématiquement dans le dossier app/http/Middleware. Dans ce composant nous aurons besoin de l'objet Session, il nous faudra donc référencer la bibliothèque Session, voici son code :

```
<?php
namespace App\Http\Middleware;
use Closure;
use Session;
class Autorisation
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if (Session::get('id') == 0) {
            return redirect('/getLogin');
        }
        return $next($request);
    }
}
```

Il n'y a pas de difficulté particulière, si l'id vaut 0, c'est que l'utilisateur ne s'est pas authentifié, on le redirige alors vers le formulaire de Login, sinon on laisse passer la requête.

Référencement

Pour qu'un MiddleWare soit actif, il faut le référencer, cela se fait dans le fichier Kernel.php où nous allons à la fois le référencer et lui associer un alias.

```
protected $routeMiddleware = [
    'autorise' => \App\Http\Middleware\Autorisation::class,
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
];
```

Utilisation

Maintenant que nous disposons d'un intercepteur actif, il faut conditionner certaines routes au fait d'être logué ou non. Il y a plusieurs façons de procéder, soit on le fait individuellement, soit on le fait collectivement.

Individuellement

```
Route::get('/uneUrl', 'UnController@uneMethode')->middleware('alias');
```

Collectivement

Cela consiste à encadrer les routes à conditionner dans un groupe activant un MiddleWare :

```
Route::group(['middleware' => ['alias']], function() {
    Route::get('/uneUrl', 'UnController@uneMethode');
    Route::get('/uneAutreUrl', 'UnAutreController@uneAutreMethode');
    . . .
});
```

Evidemment toutes les routes ne doivent pas être conditionnées 😊



Effectuer l'ensemble des opérations nécessaires et tester le bon fonctionnement du MiddleWare.