Laravel : Controllers

1. Qu'est-ce qu'un Controller ?

Dans le cadre d'un projet **Laravel**, les **Controllers** (contrôleurs) sont des classes qui vont centraliser la logique de votre application. Ils servent d'intermédiaires entre les **Routes** et les **Vues** (Views), voire directement avec la **base de données** (via les Models).

L'architecture **MVC** (Model-View-Controller) est la clé de voûte d'un projet Laravel bien organisé :

- **M** : Model (gestion des données, accès à la base de données)
- **V** : View (côté présentation, affichage des pages)
- **C** : Controller (logique métier, appels aux différents services ou Models, et renvoi des données aux Views)

---

2. Créer un Controller

Commande artisan

Pour créer un nouveau controller, on utilise la commande suivante :

```
php artisan make:controller NomController
```

Par exemple :

```
php artisan make:controller MainController
```

Cette commande va générer un fichier `MainController.php` dans le dossier `app/Http/Controllers`.

---

3. Les scénarios d'utilisation d'un Controller

3.1 Route => View directement

Parfois, on peut avoir une route qui renvoie directement une vue, sans passer par un controller :

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

Cas d'usage : une page très simple, qui n'a pas besoin de logique métier particulière.

3.2 Route => Controller (méthode) => View

Le plus souvent, on fait transiter la requête par un controller :

```
// Dans routes/web.php
Route::get('/test', [\App\Http\Controllers\MainController::class,
'index']);
```

Puis dans le `MainController`, on définit la méthode `index()` :

```
public function index()
{
    // Ici la logique de récupération de données
    // ou tout autre traitement...
    return view('test');
}
```

Ici, la route `/test` appelle la méthode `index()` de `MainController`, qui retourne la vue `test.blade.php` (dans `resources/views/test.blade.php`).

3.3 Route => Controller (méthode) => Model (Eloquent) => View

Lorsque vous devez interagir avec la base de données, la logique sera similaire, avec en plus l'appel au **Model** :

```
public function getArticles()
{
    // Récupérer tous les articles via Eloquent
    $articles = Article::all();

    // Retourner la vue "articles.blade.php"
    // et y injecter la liste des articles
    return view('articles', [
        'articles' => $articles
    ]);
}
```

Et la route associée dans `routes/web.php` :

```
Route::get('/articles', [\App\Http\Controllers\MainController::class,
'getArticles']);
```

---

4. Passer des paramètres depuis la Route vers le Controller

Supposons que vous vouliez récupérer un paramètre `id` dans votre méthode de controller. Pour cela, vous devez **définir** un paramètre dans la route :

```
Route::get('/users/{id}', [UserController::class, 'show']);
```

Dans le controller, vous pouvez accéder à ce paramètre de plusieurs manières. Par exemple, avec l'objet `Request` :

```
public function show(Request $req)
{
    $id = $req->route('id');
    // OU $id = $req->id; (Laravel 8+)

    // Logique pour récupérer et retourner l'utilisateur
    ...
}
```

---

5. Passer des paramètres depuis le Controller vers la Vue

Une fois que vous avez fait votre logique dans le controller, vous pouvez passer des **données** à la vue. Deux approches courantes :

1. **Tableau associatif en deuxième paramètre :**

   ```
   return view('nomView', [
       'cle' => $valeur,
       'autreCle' => $autreValeur
   ]);
   ```

2. **Méthode `with()` :**

   ```
   return view('nomView')
       ->with('cle1', $valeur1)
       ->with('cle2', $valeur2);
   ```

Dans la vue (`nomView.blade.php`), vous pourrez ensuite les afficher :

```
<h1>{{ $cle1 }}</h1>
<p>{{ $cle2 }}</p>
```

Exemple avec une liste d'articles

Dans votre controller :

```
public function getArticles()
{
    $articles = Article::all();

    return view('articles', [
        'articles' => $articles
    ]);
}
```

Dans la vue `articles.blade.php` :

```
<table>
    @foreach($articles as $item)
        <tr>
            <td>{{ $item->titre }}</td>
            <td>{{ $item->contenu }}</td>
        </tr>
    @endforeach
</table>
```

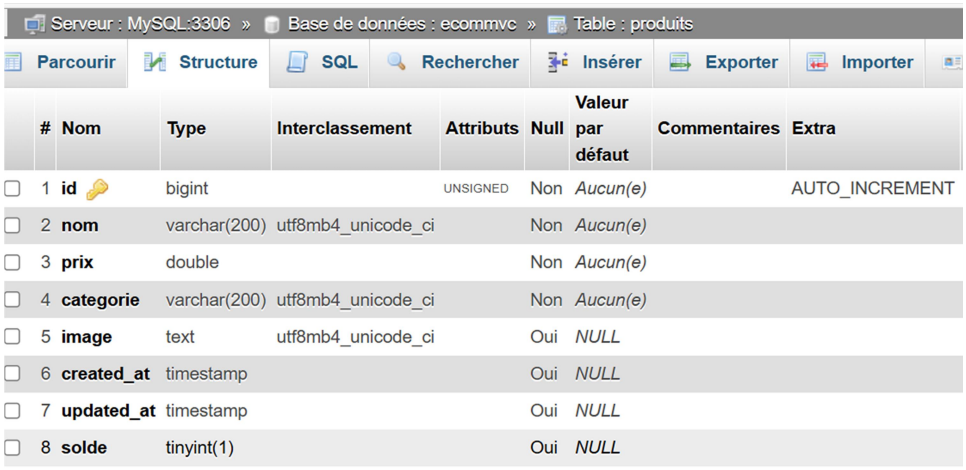
Notez qu'on peut accéder aux propriétés de `$item` via `$item->titre` si on utilise Eloquent.

---

6. Exercice : Refaire le "TP e-commerce" avec architecture MVC

Objectif

Vous avez une table `produits` dans votre base de données. Vous allez ajouter une nouvelle colonne `categorie`, puis créer une route qui permet de filtrer les produits par catégorie, et enfin afficher la liste correspondante.



#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	id	bigint		UNSIGNED	Non	*Aucun(e)*		AUTO_INCREMENT
2	nom	varchar(200)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
3	prix	double			Non	*Aucun(e)*		
4	categorie	varchar(200)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
5	image	text	utf8mb4_unicode_ci		Oui	*NULL*		
6	created_at	timestamp			Oui	*NULL*		
7	updated_at	timestamp			Oui	*NULL*		
8	solde	tinyint(1)			Oui	*NULL*		

localhost:8000/produits/electromenager

Cliquer pour avancer, maintenir pour voir l'historique

FullStack Ecom Electromenager Hicking

Liste des Prduits de la catégorie electromenager

Nom	Prix	Image
Machine à laver	3000DH	
Four	1500DH	
Micro onde	1000DH	

localhost:8000/produits/hicking

FullStack Ecom Electromenager Hicking

Liste des Prduits de la catégorie hicking

Nom	Prix	Image
sac à dos	250DH	
tente	250DH	
Montre GPS	200DH	
camping douche	400DH	

Étapes détaillées

1. **En utilisant une migration, ajouter une colonne `categorie`**
   - Créez une nouvelle migration :

     ```
     php artisan make:migration
     add_categorie_to_produits_table --table=produits
     ```

   - Dans la méthode `up()` du fichier de migration, ajoutez par exemple :

     ```
     Schema::table('produits', function (Blueprint $table) {
         $table->string('categorie')->nullable();
     });
     ```

   - Sauvegardez.
2. **Exécuter la migration**

   ```
   php artisan migrate
   ```

3. **Remplir la colonne `categorie`**
   - Vous pouvez le faire directement via Tinker, ou dans un seeder, ou en utilisant un script d'insertion. Par exemple, un **seeder** :

     ```
     php artisan make:seeder ProduitsSeeder
     ```

     Puis dans le seeder :

     ```
     public function run()
     {
         \App\Models\Produit::create([
             'nom' => 'Produit A',
             'categorie' => 'categorie1',
             // ...
         ]);
         // Créez d'autres produits selon vos besoins
     }
     ```

   - Enfin, exécutez le seeder :

     ```
     php artisan db:seed --class=ProduitsSeeder
     ```

4. **Créer une route `/produits/{cat}`**
   Dans `routes/web.php` :

   ```
   Route::get('/produits/{cat}', [ProduitController::class,
   'getProductsByCategorie']);
   ```

5. **Au niveau du Controller (`ProduitController`)**
   Créez la méthode `getProductsByCategorie($cat)` :

   ```
   public function getProductsByCategorie($cat)
   {
       // Récupérer la liste des produits dont la colonne
   'categorie' = $cat
   ```

```
    $produits = Produit::where('categorie', $cat)->get();

    // Retourner la vue 'produits' avec la liste filtrée
    return view('produits', [
        'produits' => $produits,
        'categorie' => $cat
    ]);
}
```

6. **Afficher la liste des produits dans la vue**
   Créez le fichier `resources/views/produits.blade.php` :

```
<h1>Liste des produits de la catégorie : {{ $categorie }}</h1>

<table>
    <tr>
        <th>Nom</th>
        <th>Categorie</th>
    </tr>
    @foreach($produits as $p)
        <tr>
            <td>{{ $p->nom }}</td>
            <td>{{ $p->categorie }}</td>
        </tr>
    @endforeach
</table>
```

Vous obtiendrez alors une liste de produits filtrés par catégorie, grâce à la route `/produits/{cat}`.

Conclusion

Les **Controllers** sont l'élément central pour gérer la logique et orchestrer les interactions entre les **Routes**, la **base de données** (via les **Models**), et l'**affichage** (via les **Vues**). En comprenant le fonctionnement de base, la gestion des paramètres et la manière de transmettre les données, vous serez capable de structurer votre application en **MVC** de manière propre et évolutive.

Atelier 7 :

Travail à faire

Développer et déployer en ligne la version 2 de l'application e-commerce (store en ligne) en utilisant Laravel, Vercel et AlwaysData.

À rendre :
-Rapport avec des captures d'ecran
-lien github
-lien vercel