TP n°7 - interagir avec une base de données et les formulaires avec Laravel

Site:	Université de Reims Champagne-Ardenne	Imprimé par:	CYRIL RABAT
Cours:	INFO0303 - Technologies Web 2	Date:	vendredi 13 octobre 2023, 11:37
Livre:	TP n°7 - interagir avec une base de données et les formulaires avec Laravel		

Table des matières

1. Pour commencer

Dans ce TP, nous allons utiliser *Laravel* pour interagir avec une base de données. Pour cela, nous allons créer des *migrations* et des *modèles*, ainsi que des *contrôleurs*. Une autre partie de ce TP concerne la gestion des formulaires avec la vérification automatique des données saisies. Vous pouvez réutiliser l'installation de *Laravel* que vous avez faite dans le TP précédent mais supprimez en supprimant le contrôleur `ItemController`.

2. Migrations et modèles

Dans ce chapitre, nous allons configurer *Laravel* pour accéder à une base de données et créer un modèle et une migration.

2.1. Configuration de Laravel

Dans un premier temps, il est nécessaire de spécifier la configuration de la base de données de l'application *Laravel*.

> Si le fichier `.env` n'est pas présent à la racine de votre projet *Laravel*, vous pouvez créer une copie du fichier `.env.example`. Il contient la configuration par défaut de *Laravel*.

1. Éditez le fichier `.env` et recherchez les lignes ci-dessous.

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

2. Modifiez-les en fonction de la configuration de votre SGBD et de votre base de données (normalement, vous ne devez modifier que les trois dernières lignes). Vous pouvez modifier le nom de la base (par exemple « omazone »).

Laravel est maintenant prêt pour interagir avec votre base de données.

2.2. Création d'un modèle et d'une migration

Le but d'un modèle est de pouvoir interagir plus simplement avec la base de données. *Laravel* exploite la notion de persistance des objets : le développeur manipule des objets qui sont automatiquement récupérés ou créés depuis ou dans la base de données. La création d'un modèle passe par l'outil *artisan*.

1. Créez une migration et un modèle pour les articles avec la commande suivante :

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

Un fichier `Item.php` a été créé dans le répertoire `app/models` ; il correspond au modèle. Le fichier `2023_XX_XX_XXXX_create_items_table.php` a été créé dans le répertoire `database/migrations` ; il correspond à la migration. À noter que d'autres migrations existent déjà, concernant la gestion des utilisateurs. Notez également le « s » après « item ».

Dans le fichier de migration, il y a deux méthodes : la méthode `up` qui est utilisée lors de la création ou de la mise-à-jour de la table et la méthode `down` qui est utilisée lors de l'annulation de la migration.

2. Dans la méthode `up`, modifiez l'instruction présente comme suit (pour ajouter 3 champs) :

```
Schema::create('items', function (Blueprint $table) {
      $table->id('id');
      $table->string('title', 100);
      $table->text('description');
      $table->float('price');
      $table->timestamps();
  });
```

3. Dans la méthode `down`, ajoutez ou vérifiez que le code suivant est présent (pour supprimer la table) :

```
Schema::dropIfExists('items');
```

Depuis les dernières versions de *Laravel*, une erreur peut survenir au moment d'exécuter les migrations « ... : 1071 La clé est trop longue... ». Une solution est d'éditer le fichier `config/database.php` et de rechercher les lignes suivantes :

```
'mysql' => [
...
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
...
```

Il suffit de les modifier comme suit (en supprimant "mb4") :

```
'mysql' => [
...
            'charset' => 'utf8',
            'collation' => 'utf8_unicode_ci',
...
```

4. Testez la migration en tapant la commande suivante (tapez « yes » pour créer la base) :

```
php artisan migrate
```

5. Vérifiez dans votre base de données (via *phpmyadmin*) que la table *items* a été créée (ainsi que d'autres tables).

> Dans le cas particulier de ce TP (**ce qui serait impossible pour une application en production**), il peut être nécessaire de supprimer toutes les tables et de recommencer une migration complète en cas d'erreur. Pour cela, il suffit de taper la commande suivante : `php artisan migrate:fresh`

La commande suivante permet de revenir en arrière depuis la dernière migration (cela appelle les méthodes `down`) :

```
php artisan migrate:rollback
```

6. Testez cette commande et vérifiez ses effets sur votre base de données. N'oubliez pas de refaire la migration ensuite pour recréer les tables et passer à la suite.

3. Peuplement d’une base de données

Le peuplement de la base de données consiste à la remplir avec des échantillons de données. Cela peut être nécessaire pour tester le bon fonctionnement de l’application ou pour ajouter des données de base.

3.1. Utilisation du Seeder

Le peuplement peut être effectué à l'aide d'un *seeder*. Généralement, on en crée un pour chaque table de la base de données.

1. Créez la classe `ItemSeeder` pour peupler la table *items* avec la commande suivante :

```
php artisan make:seed ItemSeeder
```

Le fichier `ItemSeeder.php` est créé dans le répertoire `database/seeds`. Le code pour le peuplement doit être placé dans la méthode `run` de la classe `ItemSeeder`.

2. Ajoutez les instructions suivantes **avant** la définition de la classe `ItemSeeder` :

```
use Illuminate\Support\Facades\DB;
use App\Models\Item;
```

3. Ajoutez le code suivant dans la méthode `run` pour créer un article (`truncate` permet de vider la table au préalable, c'est donc à utiliser avec précaution !) :

```
DB::table('items')->truncate();
Item::create([
      'title' => 'Canard en plastique',
      'description' => "Outil indispensable pour une aide au débogage.",
      'price' => '5.99'
  ]);
```

4. Exécutez le peuplement de la table avec la commande suivante :

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

5. Vérifiez dans la table *items* qu'un nouvel enregistrement a bien été créé.

Attention !!! La première instruction (`truncate`) vise à supprimer le contenu de toute la table. Si vous exécutez plusieurs fois le peuplement de la base, il y aura donc toujours le même article.

Pour éviter d'appeler les *seeders* individuellement, vous pouvez ajouter l'instruction suivante dans la méthode `run` de la classe `DatabaseSeeder` :

```
$this->call(ItemSeeder::class);
```

Cette fois-ci, pour lancer le peuplement, vous pouvez simplement saisir :

```
php artisan db:seed
```

De même, vous pouvez appeler le peuplement de la base lors de la migration :

```
php artisan migrate --seed
```

Enfin, si vous souhaitez recréer toutes les tables et lancer le peuplement de la base :

```
php artisan migrate:fresh --seed
```

6. Testez ces différentes commandes.

3.2. Utilisation d'une factory

L'utilisation d'une *factory* permet d'éviter de créer manuellement chaque élément, mais également de lever certaines vérifications lors de la création des enregistrements (c'est le cas lorsqu'il existe des liaisons entre les tables).

1. Regardez le code de la *factory* associée aux utilisateurs qui est créée par défaut lors de la création d'un projet *Laravel* : `database/factories/UserFactory`.

2. Créez une nouvelle *factory* pour les articles avec la commande suivante :

```
php artisan make:factory ItemFactory
```

3. Dans la méthode `definition`, modifiez l'instruction comme suit :

```
return [
    'title' => 'Mug Chuck Norris',
    'description' => "Vous êtes développeur ? Ce mug est fait pour vous !",
    'price' => '12.99'
  ];
```

4. Maintenant que la *factory* est prête, nous pouvons ajouter l'instruction suivante (après les précédentes) dans la méthode `run` de la classe `ItemSeeder` :

```
Item::factory()->count(1)->create();
```

5. Lancez le peuplement et vérifiez que tout fonctionne normalement. Il devrait y avoir maintenant 2 articles.

3.3. Utilisation du faker

La méthode `count` prend en paramètre le nombre d'enregistrements à créer. Mais si nous fixons une valeur supérieure à 1, tous les articles auront les mêmes données. Lorsque l'on a besoin de générer une grande quantité de données pour réaliser des tests, il est possible d'utiliser le *faker* qui génère des données aléatoires. Pour connaître les méthodes et les possibilités de cette classe, vous pouvez consultez la documentation en ligne : https://github.com/fzaninotto/Faker

1. Modifiez votre *factory* comme suit (pour le moment, seule le prix est aléatoire) :

```
return [
    'title' => 'Article bidon',
    'description' => "Ceci est un article totalement bidon !!!",
    'price' => $this->faker->randomFloat(2, 0, 99.99)
  ];
```

2. En vous aidant de la documentation, modifiez maintenant le code pour générer un intitulé aléatoire de taille 20, une description aléatoire de taille 200 et un prix aléatoire de moins de 100€.

3. Dans le *seeder*, vous pouvez maintenant spécifier plus d'enregistrements. Remplacez "1" par "10".

4. Lancez le peuplement et observez les valeurs générées dans votre base de données.

Si vous avez utilisé la méthode `realText`, vous pouvez observer que le texte est en anglais (le texte d'Alice au pays des merveilles en version originale). Vous pouvez modifier le fichier de configuration de l'application (`/config/app.php`) pour configurer le *faker* en français.

5. Cherchez la ligne `faker_locale` et remplacez `en_US` par `fr_FR`.

4. Développement d'un contrôleur

Pour le moment, nous avons une base de données qui contient une table, mais nous n'avons pas la possibilité de récupérer, d'ajouter ou de supprimer des enregistrements depuis l'application. Bien qu'il soit possible de réaliser ces opérations manuellement, nous allons utiliser un contrôleur généré automatiquement par *artisan* qui nous permettra de réaliser les actions de base CRUD (*Create*, *Read*, *Update* et *Delete*).

4.1. Création d'un contrôleur

> Si vous n'avez pas encore supprimé le contrôleur créé dans le TP précédent, supprimez-le maintenant.

1. Créez un contrôleur via *artisan* avec l'instruction suivante (à noter le `--resource` à ajouter par rapport au TP précédent) :

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

Le fichier `ItemController` a été créé dans le répertoire `app/Http/Controllers`. Il contient les méthodes classiques CRUD (vides) et quelques autres. Pour qu'il soit accessible, ajoutez la route suivante dans le fichier `routes/web.php` (supprimez la route précédente pour éviter qu'elle ne rentre en conflit) :

```
use App\Http\Controllers\ItemController;
Route::resource('item', ItemController::class);
```

2. Vérifiez que la page `item/` fonctionne (une page vide doit être affichée et non une erreur).

4.2. Affichage des articles

Pour commencer, nous allons nous intéresser à la méthode `index` du contrôleur dont le but ici, est d'afficher 10 articles.

1. Créez un *template* (fichier `template.blade.php`) (voir le code sur le dépôt *gitlab*).

2. Créez un répertoire `items` dans le répertoire des vues (`ressources/views/`) et à l'intérieur, créez une vue pour lister les articles (fichier `list.blade.php`) (voir le code sur le dépôt *gitlab*).

3. Spécifiez le code suivant dans la méthode `index` du contrôleur :

```
$itemList = Item::orderBy('title', 'desc')->take(10)->get();
return view('items.list', ['itemList' => $itemList]);
```

> Attention !!! La classe `Item` n'est pas reconnue. N'oubliez pas de spécifier le *namespace* ! On ne le précisera plus dans la suite.

La première instruction vise à récupérer les 10 derniers articles. Pour cela, nous utilisons la méthode `orderBy` qui permet de spécifier le tri (ici sur le champ *intitulé* avec un tri décroissant) et la méthode `take` qui indique le nombre d'éléments attendus (ici 10). La deuxième instruction appelle la vue en passant en paramètre la liste des articles à afficher. Le nom de la vue est *items.list* car elle se situe dans le répertoire `items`.

4. Vérifiez que la page `items/` affiche bien la liste des articles.

À côté de chaque article, un bouton avec un oeil permet de consulter un article en particulier. Pour le moment, il n'est pas fonctionnel.

5. Créez une vue pour afficher un article (fichier `show.blade.php` dans le répertoire `items`) en récupérant le code sur le dépôt *gitlab*.

6. Modifiez la méthode `show` du contrôleur comme suit :

```
return view('items.show', ['item' => Item::findOrFail($id)]);
```

La méthode `findOrFail` permet de récupérer l'article dont l'identifiant est `id` (passé via l'URL). Si l'identifiant est invalide, *Laravel* fait automatiquement une redirection sur la page d'erreur. Le lien vers un article spécifique s'écrit de la manière suivante (avec *blade*) :

```
{{route('items.show', $item->id)}}
```

7. Vérifiez maintenant qu'il est possible d'afficher le détail d'un article en cliquant sur le bouton.

4.3. Ajout d'un article

Pour ajouter un nouvel article, nous avons besoin de créer une vue qui va contenir un formulaire, permettant à l'utilisateur de saisir les informations.

1. Ajoutez le code suivant dans la vue `item.list` (avant la balise ul) :

```
<div class="d-flex justify-content-center">
  <a href="{{route('item.create')}}" class="btn btn-sm btn-primary mb-1">
    Création d'un nouvel article
  </a>
</div>
```

2. Créez une vue pour la création d'un nouvel article (fichier `create.blade.php`) (voir le code sur le dépôt *gitlab*).

3. Dans la méthode `create` du contrôleur ajoutez le code suivant :

```
return view('items.create');
```

4. Vérifiez le bon fonctionnement de l'URL `item/create` en cliquant sur le bouton que nous venons d'ajouter. Le formulaire doit s'afficher.

5. Que se passe-t-il lorsque vous cliquez sur le bouton « Ajouter » ?

Lavarel gère automatiquement le rejeu des formulaires. Par défaut, vous obtenez l'erreur 419 pour « Page expirée ». Il faut ajouter un champ spécifique dans chaque formulaire pour qu'il soit considéré comme valide par *Laravel*.

6. Ajoutez l'annotation *blade* `@csrf` après la balise `form` du formulaire. Vérifiez que l'erreur n'est plus affichée.

Pour le moment, l'article n'est pas ajouté dans la base. Lorsque le formulaire est validé, la méthode `store` du contrôleur est appelée automatiquement.

7. Ajoutez le code suivant (ainsi que les instructions `use` nécessaires avant la classe `ItemController`) pour récupérer l'article saisi et le sauvegarder dans la base (pour le moment, nous ne faisons pas de vérifications) :

```
$item = new Item();
$item->title = $request->title;
$item->description = $request->description;
$item->price = floatval($request->price);
$item->save();

return redirect()->route('item.show', ['item' => $item]);
```

8. Testez maintenant l'ajout d'un article.

Nous avons fait le choix de récupérer les champs un par un. Mais il est possible de récupérer automatiquement tous les champs. Pour cela, il faut modifier le modèle `Item.php` (répertoire `app/Models`) et spécifier les champs qui peuvent être remplis.

9. Ajoutez le code suivant dans la classe `Item.php` :

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

10. Remplacez le code de la méthode `store` du contrôleur par le code suivant :

```
$item = Item::create($request->input());
$item->save();
return redirect()->route('item.show', ['item' => $item]);
```

11. Vérifiez que le fonctionnement est identique.

4.4. Validation des données

Pour le moment, l'ajout d'un article peut générer une erreur si l'utilisateur omet le titre, la description ou le prix. Nous allons ajouter la vérification des données saisies dans le contrôleur.

1. Au début de la méthode `store` du contrôleur, ajoutez le code suivant :

```
$validated = $request->validate([
    'title' => ['required', 'max:100'],
    'description' => ['required'],
    'price' => ['required', 'numeric', 'min:0']
  ]);
```

Maintenant, si l'utilisateur oublie de saisir des données, il est redirigé automatiquement sur le formulaire de création... sans message d'erreur !

2. Ajoutez le code suivant dans la vue `items.create`, avant le formulaire :

```
@if($errors->any())
  <div class="alert alert-danger">
    <ul class="mb-0">
      @foreach($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif
```

3. Essayez de créer un article sans remplir les champs du formulaire et observez l'affichage.

4. Pour identifier les champs invalides, vous pouvez ajouter le code suivant dans l'attribut `class` de chaque champ du formulaire (ici pour le champ *title*) :

```
@error('title') is-invalid @enderror
```

Pour finir, il peut être intéressant que le formulaire soit prérempli avec les données précédemment saisies pour éviter que l'utilisateur ne remplisse tout à nouveau en cas d'erreur. Pour cela, il suffit d'ajouter l'attribut `value` (pour les champs texte) et de récupérer la valeur saisie avec la fonction `old`.

5. Ajoutez l'attribut suivant dans le champ correspondant au titre (et faites de même pour les autres champs) :

```
value="{{ old('title') }}"
```

Comme nous pouvons le constater, les messages d'erreur sont affichés en anglais. Par défaut, *Laravel* utilise la langue *en* correspondant au répertoire `lang/en`.

6. Commencez par récupérer les fichiers de langue avec les commandes suivantes (à la racine de l'application) :

```
composer require laravel-lang/common --dev
php artisan lang:add fr
php artisan lang:update
```

Dans le répertoire `vendor/laravel-lang/lang/locale/`, vous trouverez les fichiers pour un grand nombre de langues.

7. Modifiez la langue par défaut de *Laravel* en modifiant le fichier `config/app.php` (recherchez `local` et remplacez *en* par *fr*).

8. Vérifiez que les messages d'erreur sont maintenant en français.

Dans le fichier `lang/fr/validation.php`, le tableau retourné contient une entrée *attributes* correspondant à un tableau. Si vous avez des noms de champ qui ne sont pas en français, vous pouvez en ajouter de nouveaux (ici, tous les champs utilisés sont présents).

4.5. Personnalisation des messages d'erreur

Grâce à la section précédente, les messages sont bien affichés en français. Cependant, ils ne sont pas encore parfaits. Par exemple, si on omet de spécifier le titre, le message affiché est le suivant :

```
Le champ titre est obligatoire
```

Au moment de l'appel à la méthode `validate`, il est possible de spécifier, en plus des règles de validation, les messages associés.

1. Remplacez la première instruction de la méthode `store` par les suivantes (les variables `$fields` et `$msgs` ne sont pas obligatoires, mais elles permettent une meilleure visibilité) :

```
$rules = [
        'title' => ['required', 'max:100'],
        'description' => ['required'],
        'price' => ['required', 'min:0', 'numeric']
    ];

$msgs = [
        'title.required' => 'Il faut spécifier un intitulé',
        'title.max' => 'L\'intitulé ne doit pas contenir plus de 100 caractères',
        'description.required' => 'Il faut spécifier une description',
        'price.required' => 'Il faut spécifier un prix',
        'price.numeric' => 'Le prix est incorrect',
        'price.min' => 'Le prix est obligatoirement positif'
    ];

$validated = $request->validate($rules, $msgs);
```

2. Vérifiez que les messages d'erreur sont différents. Testez tous les cas.

Le tableau `$msgs` contient tous les messages personnalisés en fonction du champ et du type d'erreur rencontrés. Par exemple, si le titre n'est pas spécifié, le message d'erreur associé est `title.required`.

Pour simplifier l'écriture de cette méthode, nous pouvons également créer une classe `Request` spécifique. Tout d'abord, nous utilisons *artisan* pour créer une classe qui hérite de la classe `FormRequest`.

3. Exécutez la commande suivante :

```
php artisan make:request StoreItemRequest
```

Une classe `StoreItemRequest` est créée dans le répertoire `app/Http/Requests/`. Par défaut, elle contient deux méthodes. La méthode `authorize` retourne `false` par défaut, ce qui veut dire que l'utilisateur n'a pas les droits.

4. Remplacez `false` par `true` dans la méthode `authorize` (nous n'avons pas de gestion des utilisateurs pour le moment).

La méthode `rules` correspond au tableau de règles de validation : elle retourne simplement le contenu de la variable `$rules` précédente.

5. Modifiez la méthode `rules`.

De la même manière, nous pouvons ajouter la méthode `messages` pour les messages personnalisés et la méthode `attributes` pour personnaliser les noms des attributs.

6. Ajoutez la méthode `messages` dont le code est le suivant :

```
public function messages()
{
    return [
            'title.required' => 'Il faut spécifier un intitulé',
            'title.max' => 'L\'intitulé ne doit pas contenir plus de 100 caractères',
            'description.required' => 'Il faut spécifier une description',
            'price.required' => 'Il faut spécifier un prix',
            'price.numeric' => 'Le prix est incorrect',
            'price.min' => 'Le prix est obligatoirement positif'
        ];
}
```

7. Ajoutez la méthode `attributes` dont le code est le suivant :

```
public function attributes()
    {
        return [
                'title' => 'title',
                'description' => 'description',
                'price' => 'price'
            ];
    }
```

8. Remplacez le code de la méthode `store` du contrôleur par le code suivant (n'oubliez pas de modifier le paramètre et de mettre la bonne instruction `use` avant votre classe) :

```
public function store(StoreItemRequest $request)
    {
        $request->validated();
        $item = Item::create($request->input());
        $item->save();

        return redirect()->route('item.show', ['item' => $item]);
    }
```

9. Vérifiez que tout fonctionne comme avant.

4.6. Édition des articles

Nous allons maintenant ajouter la possibilité de modifier une actualité. Pour cela, il faut ajouter un nouveau bouton dans la liste des actualités, créer une vue et modifiez le contrôleur.

1. Ajoutez le code suivant dans la vue `items.list` après le premier bouton :

```
<a href="{{route('item.edit',$item->id)}}" class="btn btn-sm btn-primary mb-1">
  <i class="bi bi-pencil-square"></i>
</a>
```

2. Créez la vue pour éditer une actualité (fichier `edit.blade.php`) (voir le fichier sur le dépôt *gitlab*).

Dans ce fichier, vous pouvez remarquer l'annotation `@method('PUT')` dans le champ du formulaire. Cela permet au contrôleur de savoir quelle méthode appeler lors de la validation du formulaire.

3. Spécifiez le code suivant dans la méthode `edit` du contrôleur :

```
return view('items.edit', ['item' => Item::findOrFail($id)]);
```

Lorsque le formulaire est validé, c'est la méthode `update` du contrôleur qui est appelée.

4. Spécifiez le code suivant dans la méthode `update` (n'oubliez pas de modifier l'en-tête) :

```
public function update(StoreItemRequest $request, Item $item)
{
    $request->validated();
    $item->update($request->input());
    return redirect()->route('item.show', ['item' => $item]);
}
```

5. Vérifiez que l'édition est fonctionnelle.

4.7. Suppression des articles

Pour finir, nous souhaiterions pouvoir supprimer des articles. Pour cela, il faut utiliser un formulaire pour spécifier la méthode `DELETE` au contrôleur à l'aide de l'annotation *blade* `@method('DELETE')` (sans oublier le champ `@csrf`).

1. Ajoutez le formulaire suivant après la balise dans la vue `items.list` :

```
<form id="deleteForm" action="" method="POST">
  @method('DELETE')
  @csrf
</form>
```

2. Ajoutez le bouton suivant après les deux autres boutons :

```
<button type="submit" formaction="{{route('item.destroy', $item->id)}}" form="deleteForm" class="btn btn-sm btn-danger mb-1">
  <i class="bi bi-trash"></i>
</button>
```

3. Spécifiez le contenu de la méthode `destroy` du contrôleur :

```
$item = Item::findOrFail($id);
$item->delete();
return redirect()->route('item.index');
```

4. Vérifiez que la suppression est opérationnelle.

5. Annexes : commandes Laravel

Installation d'une application *Laravel* de zéro :

```
composer create-project --prefer-dist laravel/laravel exemple
```

Installation d'une application *Laravel* existante (récupérée depuis un dépôt, par exemple) :

```
composer install
php artisan key:generate
```

Création du lien pour les ressources publiques (images, CSS, JavaScript) :

```
php artisan storage:link
```

Installation de la barre de debug :

```
composer require barryvdh/laravel-debugbar --dev
```

Publication d'un package "vendor" :

```
php artisan vendor:publish
```

Liste des routes :

```
php artisan route:list
```

Lancement de la migration :

```
php artisan migrate
```

Annulation de la dernière migration :

```
php artisan migrate:rollback
```

Lancement de la migration ET du peuplement de la base :

```
php artisan migrate --seed
```

Lancement de la migration depuis zéro (suppression des tables et des données) :

```
php artisan migrate:fresh
```

Lancement de la migration depuis zéro ET du peuplement de la base :

```
php artisan migrate:fresh --seed
```

Création d'une migration (en spécifiant la table associée) :

```
php artisan make:migration nom_migration --table=nom_table
```

Création d'un modèle et de la migration associée :

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

Création d'un seeder :

```
php artisan make:seed nom_seeder
```

Création d'une factory :

```
php artisan make:factory nom_factory
```

Lancement du peuplement de la base :

```
php artisan db:seed
```

Lancement du peuplement de la base d'un seeder spécifique :

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

Création d'un contrôleur :

```
php artisan make:controller nom_controller
```

Création d'un contrôleur CRUD :

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

Création d'une requête :

```
php artisan make:request nom_request
```