Laravel : Quelques commandes artisan:

1. Introduction

Laravel Artisan est une Interface en Ligne de Commande (CLI) qui va vous permettre de gérer votre application en lançant des commandes via le terminal. Cette commande vous permettra d'effacer le cache de l'application, gérer des modèles, des contrôleurs, des routes… Ces commandes vont permettre de vous faire gagner du temps lors du développement de votre application Web.

Pour afficher une liste de toutes les commandes Artisan disponibles, vous pouvez utiliser la commande **list** :

```
php artisan list
```

Toutes les routes définies :

```
php artisan route:list
```

a. Laravel Tinker (Repl)

Laravel Tinker vous permet d'interagir avec une base de données sans créer les routes. Laravel **tinker** est utilisé avec un artisan php pour créer les objets ou modifier les données. Le php artisan est une interface de ligne de commande qui est disponible avec un Laravel. Un tinker joue autour de la base de données, c'est-à-dire qu'il vous permet de créer les objets, d'insérer les données, etc.

Installation

Toutes les applications Laravel incluent Tinker par défaut. Cependant, vous pouvez installer Tinker en utilisant Composer si vous l'avez préalablement supprimé de votre application :

```
composer require laravel/tinker
```

Pour entrer dans l'environnement Tinker, exécutez la commande donnée ci-dessous :

```
php artisan tinker
```

```
PS C:\xampp\htdocs\tp1> php artisan tinker
Psy Shell v0.11.9 (PHP 8.1.12 — cli) by Justin Hileman
> Student::find(4)
[!] Aliasing 'Student' to 'App\Models\Student' for this Tinker session.
= null

> Student::find(74)
= App\Models\Student {#4736
    id: 74,
    name: "Sara123",
    picture: "students/38Es1HLrkzCtox9oYRKYwtiab1DPLiBSE9voypf4.png",
    section: "info",
    mail: "sara@gmail.com",
    created_at: "2023-01-31 21:29:27",
    updated_at: "2023-02-01 09:01:38",
  }
```

Nous pouvons créer les enregistrements dans les tables de la base de données en utilisant l'outil de ligne de commande. Nous utilisons l'instruction suivante dans l'outil de ligne de commande qui insère les données directement dans la table de la base de données :

```
Post::create(["title"=>"new post","content"=>"comment for this post"])
```

```
PS C:\xampp\htdocs\tp1> php artisan tinker
Psy Shell v0.11.9 (PHP 8.1.12 — cli) by Justin Hileman
> Post::create(["title"=>"new post","content"=>"comment for this post"])
[!] Aliasing 'Post' to 'App\Models\Post' for this Tinker session.
= App\Models\Post {#4730
    title: "new post",
    content: "comment for this post",
    updated_at: "2023-02-12 17:44:07",
    created_at: "2023-02-12 17:44:07",
    id: 3,
  }
```

Nous pouvons récupérer les enregistrements de la base de données de trois façons :

La première façon est d'utiliser la méthode **find().**

```
Student::find(74)
```

```
Post::where('id',1)->first() //Récupérer un seul enregistrement
```

```
Post::where('id','>',1)->get()
```

Dans l'écran ci-dessus, nous récupérons les enregistrements dont l'identifiant est supérieur à 1. Dans ce cas, plus d'un enregistrement est récupéré, nous utilisons donc la méthode **get()**. Comme la méthode **get()** est utilisée lorsqu'un tableau d'enregistrements est récupéré.

2. Comment créer une commande artisan personnalisée dans Laravel

Laravel est un framework complet qui offre de nombreuses commandes **artisan** pour automatiser diverses actions, comme la création d'un contrôleur, la population de la base de données et le démarrage du serveur. Cependant, lorsque vous construisez des solutions personnalisées, vous avez vos propres besoins spécifiques, qui peuvent inclure une nouvelle commande. Laravel ne vous limite pas à ses seules commandes ; vous pouvez créer les vôtres en quelques étapes.

Voici les étapes à suivre pour créer une nouvelle commande artisanale.

Étape 1 : Créer une commande

Utilisez la commande **make:command** pour créer une nouvelle commande. Il suffit d'entrer le nom de la commande, comme ceci :

```
php artisan make:command CreatePostCommand
```

Dans cet exemple, nous allons créer une commande appelée **CreatePostCommand**.

La commande crée un fichier **CreatePostCommand**.php, nommé d'après le nom de la commande, dans un répertoire **Commands** nouvellement créé dans le dossier Console.

Le fichier généré contient les configurations de la commande nouvellement créée qui sont faciles à comprendre et à modifier.

Étape 2 : Personnaliser la commande

Tout d'abord, définissez la signature de la commande. C'est ce qui sera mis après php artisan pour exécuter la commande. Dans cet exemple, nous utiliserons check:posts, donc la commande sera accessible en exécutant :

```
php artisan create:post
```

Pour ce faire, mettez à jour la propriété **$signature** de la commande, comme ceci :

```
protected $signature = 'create:post' ;
```

Ensuite, configurez une description appropriée qui s'afficherait lorsque la liste php artisan affiche la commande avec d'autres commandes.

Pour ce faire, mettez à jour la propriété **$description** pour qu'elle corresponde à ceci :

```
protected $description = 'Creates new Post';
```

Enfin, dans la méthode **handle(),** effectuez l'action que vous souhaitez qu'elle effectue. Dans cet exemple, le nombre de post sur la plateforme est renvoyé.

```
public function handle()
{
    // add a random post to database
}
```

Étape 3 : Test de la commande

Dans le terminal, exécutez la commande pour intégrer un nouvel enregistrement dans votre base de données.

```
php artisan create:post
```

Passage d'arguments à la commande

Vous pouvez avoir une commande qui nécessite un argument pour la fonction. Par exemple, une commande pour effacer tous les commentaires d'un post spécifique de la base de données nécessiterait l'identifiant du post.

Pour ajouter un argument, mettez à jour la chaîne $signature et ajoutez l'argument entre accolades.

```
protected $signature = 'remove:comments {postId}' ;
```

Cette commande serait alors appelée lorsque l'id d'un post est 5 :

```
php artisan remove:comments 5
```

À d'autres moments, vous voulez pouvoir passer un argument, mais pas toujours, alors vous pouvez rendre l'argument facultatif en ajoutant un point d'interrogation à la fin, comme suit :

```
protected $signature = 'remove:comments {postId?}' ;
```

Vous pouvez également définir une valeur par défaut pour un argument, comme suit :

```
protected $signature = 'remove:comments {postId=6}' ;
```

Vous pouvez également passer plusieurs arguments et les rendre facultatifs ou avec des valeurs par défaut comme vous le souhaitez.

`protected $signature = 'remove:comments {postId} {$numbers_comments?}'`

On peut accéder à ces arguments en utilisant :

`$this->arguments()`

Cela renvoie un tableau associatif avec les arguments comme clé et leurs valeurs comme valeurs. Ainsi, pour accéder à l'argument **postId**, vous pouvez l'obtenir comme ceci :

`$post_id = $this->arguments()['postId'] ;`

Cependant, il existe une autre façon d'obtenir un seul argument :

`$post_id = $this->argument('postId') ;`

```php
<?php

namespace App\Console\Commands;

use App\Models\Post;
use Illuminate\Console\Command;

class CountPosts extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'remove:comments {postId}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'remove comments for given posId';

    /**
     * Execute the console command.
     *
     * @return int
     */
```

```
    public function handle()
    {
        $postId=$this->argument('postId');
        $post=Post::find($postId);
        $post->comments()->delete();
    }
}
```

Dans le terminal, exécutez la commande pour supprimer tous les commentaires du post ayant id=10.

```
php artisan remove:comments 10
```

Passer des options à la commande

Les options, comme les arguments, sont une autre forme de saisie de l'utilisateur. Les options sont préfixées par deux traits d'union (--) lorsqu'elles sont fournies par la ligne de commande. Il existe deux types d'options : celles qui reçoivent une valeur et celles qui n'en reçoivent pas. Les options qui ne reçoivent pas de valeur servent de "commutateur" booléen. Voyons un exemple de ce type d'option :

Par exemple, pour obtenir le nombre d'utilisateurs dont l'adresse électronique est vérifiée, vous pouvez passer une option **--verified** à la commande. Pour créer une option, passez-la dans la propriété **$signature** comme l'argument, mais préfixez-la par **--**.

```
protected $signature = 'check:users {--verified}' ;
```

Maintenant, la commande peut être utilisée avec une option comme ceci :

```
php artisan check:users --verified
```

Dans cet exemple, l'option **--verified** peut être spécifié lors de l'appel de la commande Artisan. Si le booléen **--verified** est passé, la valeur de l'option sera **true**. Sinon, la valeur sera **false**.

Vous pouvez définir une valeur par défaut pour une option ou la définir pour exiger une valeur.

```
protected $signature = 'check:users {--verified} {--add=} {--delete=5}' ;
```

Dans cet exemple, **add** nécessite une valeur pour être utilisé et **delete** a une valeur par défaut de 5. **verified** se voit attribuer une valeur booléenne selon qu'elle est passée ou non.

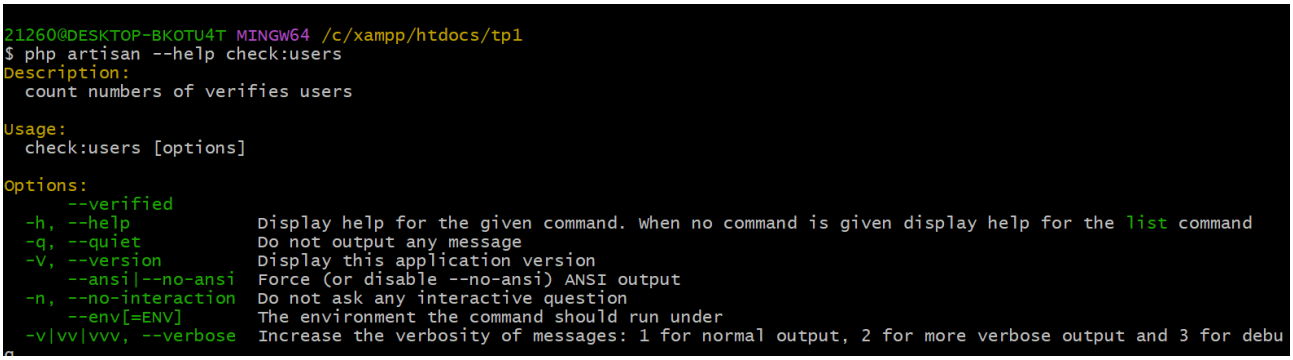
On peut accéder facilement à la valeur de ces options en utilisant **$this->option('verified')** pour les uniques et **$this->options()** pour obtenir toutes les options sous forme de tableau associatif.

Décrire les paramètres d'entrée

Jusqu'à présent, nous avons appris à accepter des arguments et même des options, mais lorsque la commande est utilisée avec l'aide de php artisan, ces entrées n'ont pas de description. Pour les définir, il suffit d'ajouter un deux-points, **:**, après le nom de l'argument ou de l'option.

```
Protégé $signature = '
        check:users
        {userId : Id de l'utilisateur à récupérer}
        {--verified : Obtient le nombre des utilisateurs vérifiés}
    ' ;
```

Maintenant, lorsque **php artisan --help check:users** est exécuté, vous devriez voir quelque chose comme ceci :

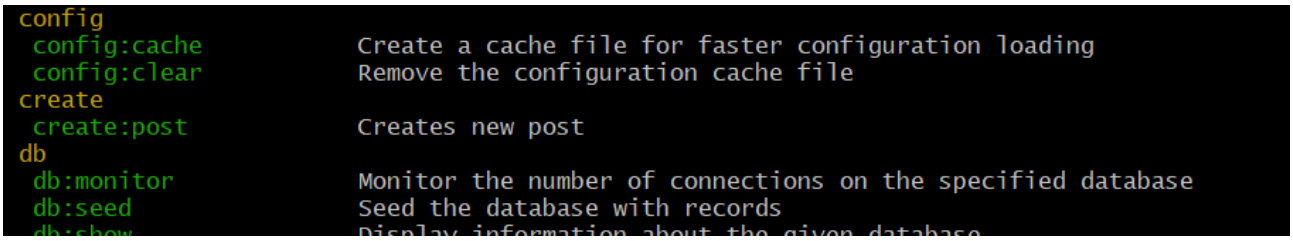


Exercice :

1- Créer une commande **CreatePostCommand** qui permet d'ajouter un nouvel post.

2- Personnalisez les propriétés **$description** et **$signature**.

3- Afficher la liste des commandes avec **php artisan list.**



4- Tester la commande.

5- Ajouter un post via les arguments et les options