Laravel - Blog APP: CRUD - Part 1

1 - Création d'une nouvelle application Laravel

Ouvrez votre console et cd dans le répertoire www dans votre répertoire d'installation MAMP, LAMP ou WAMP.

Tapez la commande suivante :

```
composer create-project --prefer-dist laravel/laravel my-blog “8.*”
```

Il créera un répertoire appelé my-blog et y chargera tous les fichiers principaux de Laravel.

---

2 - Configuration de l'application Laravel

Après avoir installé l'application Laravel, nous devrons configurer notre base de données pour qu'elle fonctionne.

En utilisant PhpmyAdmin ou workbench, créez une nouvelle base de données nommée my_blog

```
CREATE DATABASE my_blog
```

Ouvrez votre explorateur de fichiers et accédez au dossier mon-blog

Ouvrez le fichier .env dans votre éditeur de code

Modifiez les éléments suivants dans le fichier : -

Clé APP_NAME pour le nom de votre blog, c'est-à-dire « Mon blog »

Clé DB_DATABASE vers le nom de la base de données, c'est-à-dire my_blog

Le fichier final devrait ressembler à ceci :

```
...
APP_NAME="My Blog"
...
DB_CONNECTION=
mysqlDB_HOST=
127.0.0.1
DB_PORT=3306DB_DATABASE=
my_blogDB_USERNAME=
rootDB_PASSWORD=
...
```

Avec tout configuré, il est temps d'exécuter notre application et de voir à quoi elle ressemble.

Pour exécuter l'application, tapez la commande suivante :

```
php artisan serve
```

Il démarrera l'application et vous donnera une URL, http://127.0.0.1:8000, ouvrez l'URL dans votre navigateur et voyez l'application

---

3 - La structure du fichier de l'application Laravel

Dossier **app** - Il contient toute la logique de notre application, cela inclut les modèles, les contrôleurs, les fournisseurs de services, etc.

Dossier **models** - C'est là que la logique métier de votre application est stockée, un modèle est une représentation d'un objet de la vie réelle. Par exemple, un article de blog.

Les modèles seront générés à l'aide de la commande php artisan make:model et suivront une convention de libellé de casse de titre au singulier. Par exemple, pour un modèle de publication de blog, nous pourrions l'appeler BlogPost.php.

***Remarque :** Laravel est livré avec un modèle User.php prêt à l'emploi qui définit les détails de l'utilisateur :

Dossier **Http/Controllers** – Il contiendra tous les fichiers de contrôleur de votre application.

Un contrôleur crée un lien entre vos Modèles et vos Vues. Lorsqu'un nouveau formulaire d'article de blog est soumis par l'utilisateur, les données entrent dans le contrôleur où elles sont nettoyées, puis transmises au modèle pour être stockées dans la base de données, puis le contrôleur renvoie des commentaires à la vue indiquant que l'article de blog a été créé. .

Les contrôleurs seront générés à l'aide de la commande php artisan make:controller et suivront une convention de libellé de casse de titre au singulier avec le mot Controller à la fin. Pour notre contrôleur de publication de blog, nous l'appellerons BlogPostController.php

Un contrôleur dispose de 7 méthodes de signature qui permettent des opérations brutes :

1. **index ()** - pour récupérer toutes les ressources, par exemple tous les articles de blog disponibles.
2. **show()** - pour récupérer une seule ressource, par ex. un seul article de blog, disons, l'article 5.
3. **create()** - affiche le formulaire à utiliser pour créer une ressource (non disponible pour les contrôleurs API).
4. **store()** - pour valider la ressource dans la base de données, par ex. enregistrer l'article de blog.
5. **edit()** - pour afficher le formulaire pour modifier la ressource (non disponible pour les contrôleurs API).
6. **update()** - pour valider la ressource éditée dans la base de données.
7. **destroy()** - pour supprimer une ressource de la base de données.

Revenons maintenant à notre structure de fichiers d'application Laravel :

Dossier **Http/Middleware** - Il contient tous les middleware, le middleware est le code qui doit être exécuté avant que la demande n'atteigne le contrôleur, par ex. Authentifier un utilisateur avant d'autoriser l'accès.

Dossier **Exceptions** - Il contient toute la gestion des exceptions dans votre application, vous pouvez également ajouter des exceptions personnalisées ici.

Dossier **console** - Il contient toutes les commandes PHP artisan (PHP Artisan est l'outil de ligne de commande fourni avec Laravel pour nous aider à concevoir notre application plus rapidement). Ces commandes sont utilisées pour créer des fichiers d'application et également effectuer certaines actions comme démarrer un serveur de développement. Un exemple de commande artisan est celle que nous avons exécutée au début après l'installation de Laravel (php artisan serve).

Dossier **providers** - Il contient tous les fournisseurs de services de votre application, un fournisseur de services dans Laravel est un groupe de code qui effectue une tâche spécifique dans l'application chaque fois que nécessaire. Par exemple, un fournisseur de services de facturation sera conçu pour autoriser plusieurs plates-formes de paiement, mais tout ce que vous avez à faire est d'appeler le fournisseur de services et il fournira dynamiquement une plate-forme de paiement au lieu de spécifier une plate-forme dans le contrôleur.

Dossier **bootstrap** - Il contient le fichier d'application qui amorce le framework en l'initialisant

(configuration du chemin et de l'environnement), il contient également le dossier de cache qui contient les fichiers générés par le framework pour l'optimisation de l'application.

***REMARQUE :** le dossier bootstrap n'a rien à voir avec Bootstrap CSS Framework.

Dossier **config** - Il contient tous les fichiers de configuration de l'application. Pour obtenir une certaine configuration, Laravel fournit une méthode d'assistance pour le faire.

Par exemple:

Obtenir le nom de l'application que nous utiliserions :

```
config('app.name', 'Default Name')
```

Dans cet exemple, app est le fichier de configuration dans lequel nous recherchons, name est la clé tandis que « Default Name » est le nom qui sera utilisé au cas où la clé ou le fichier n'existerait pas.

Dossier **database** – Ce dossier contient les migrations de base de données, les usines et les semences. Les migrations sont des définitions de tables de base de données telles que les colonnes et leurs types de données, les définitions de clés nécessaires, etc.

Les usines sont des plans utilisés pour créer des exemples de données pour la base de données tandis que les graines sont les exemples de données pour notre base de données. Ce sont en fait des commandes qui déclenchent la création d'échantillons de données lorsqu'elles sont exécutées.

Vous pouvez également choisir de stocker les fichiers de base de données SQLite ici.

***Remarque :** Laravel est livré avec la migration users_table et l'usine UserFactory.php prête à l'emploi qui aideront à créer une table d'utilisateurs et à définir des exemples de données pour notre table d'utilisateurs.

Dossier **public** - Ce dossier contient le fichier d'index qui est le point d'entrée de l'application, une fois la demande effectuée, il atteint ce fichier et est ensuite dirigé vers la route nécessaire. Vous pouvez également stocker des ressources publiques ici, telles que des images publiques, des css et des js.

Dossier **resources** - Ce dossier contient les fichiers sources compatibles de notre application, notamment les fichiers vues, sass et JavaScript brut (principalement Node.js ou à partir de JS Frameworks). Les vues sont créées à l'aide de HTML avec un mélange d'un moteur de modélisation Laravel appelé blade, nous en apprendrons plus à ce sujet plus tard.

Dossier **routes** - Ce dossier contient tous les fichiers de route vers notre application, ces fichiers de route incluent web.php, api.php, channels.php, console.php. Chaque fichier contient plusieurs itinéraires définis par l'utilisateur. Une route est simplement une adresse Web qui pointe vers une certaine fonction soit dans le fichier de routes, soit dans le contrôleur.

Dossier **storage** - Il contient tous les fichiers privés, tels que les images de profil client. Un lien dynamique peut être créé à partir d'ici vers le public. Tous les journaux d'applications sont également stockés ici.

Dossier **tests** - C'est là que vos tests d'applications sont stockés.

Dossier **vendor** – C'est là que tous les packages tiers apportés par composer sont stockés.

Fichier **.env** – Ce fichier contient les variables d'environnement que ces variables sont importées via les fichiers de configuration à l'aide de la méthode d'assistance env.

---

4 - Faire le modèle BlogPost

Tout d'abord, nous allons créer un modèle BlogPost, pour créer un modèle, nous utilisons la commande php artisan make:model suivie du nom du modèle.

```
php artisan make:model BlogPost
```

Cela créera un fichier appelé BlogPost.php dans notre dossier App/Models, et mesdames et messieurs, c'est tout ce que vous devez faire pour créer un modèle.

---

5 - Migration de la table blog_posts et migration de la base de données

Pour créer une migration, nous utilisons la commande php artisan make:migration suivie des mots action_table_name_table.

Dans notre cas:

从 Laravel建数据表， 执行migrate迁移命令将在phpmyadmin生成真实的表

```
php artisan make:migration create_blog_posts_table
```

***ASTUCE :** assurez-vous toujours que le nom de votre table est au pluriel du nom de votre

modèle en minuscules.

Cela créera un fichier dans le dossier base de données/migrations. Le fichier aura l'horodatage actuel précédant le nom que vous avez donné dans la commande : 2020_11_17_163409_create_blog_posts_table.php.

Une fois que vous avez créé la migration, nous devons la remplir avec les champs dont nous avons besoin dans la méthode Schema::create, notre fichier final ressemblera à ceci :

```
    public function up
()
    {        Schema::create('blog_posts', function (Blueprint
 $table) {            $table->id
();
          /* Nous avons commencé à ajouter du code ici*/
            $table->text('title');
// Titre de notre article de blog                        $table->text(
'body');
// Corps de notre article de blog
            $table->text('user_id');
// user_id de l'auteur de notre article de blog
          /* Nous avons arrêté d'ajouter du code ici*/
            $table->timestamps
();
        });
```

$table->id(); - Crée un champ ID qui est également la clé primaire de notre table.

$table->timestamps(); - Crée deux champs TIMESTAMP (created_at & updated_at).

Migration Types de données:

https://laravel.com/docs/8.x/migrations#available-column-types

Migration Contraintes:

Command	Description
`$table->primary('id');`	Ajoute une clé primaire.
`$table->primary(['id', 'parent_id']);`	Ajoute des clés composites.
`$table->unique('email');`	Ajoute un index unique.
`$table->index('state');`	Ajoute un index.

Command	Description
`$table->spatialIndex('location');`	Ajoute un index spatial (sauf SQLite).
`$table->foreign('user_id')->references('id')->on('users');`	Ajoute une contrainte de clé étrangère

https://laravel.com/docs/8.x/migrations#available-index-types

Après avoir créé la migration, tout ce que nous avons à faire est de migrer pour créer les tables dans notre base de données. Pour migrer, exécutez la commande de migration (ci-dessous).

Cela créera des tables dans la base de données :

如果生成表的过程出现错误提示
https://stackoverflow.com/questions/42244541/laravel-migration-error-syntax-error-or-access-violation-1071-specified-key-wa

```
php artisan migrate
```

Update your /app/Providers/AppServiceProvider.php to contain:

---

6 - Création d'une usine et d'une graine pour notre table blog_post

Maintenant que nous avons créé notre table, il est temps de la remplir avec des données. La fabrique d'utilisateurs (UserFactory.php) existe déjà et nous allons maintenant créer une fabrique pour notre article de blog.

Pour créer une usine, nous utilisons la commande make:factory suivie du nom de la classe, nous ajoutons également l'indicateur -m suivi du nom du modèle pour attribuer une usine à ce modèle.

```
php artisan make:factory BlogPostFactory –m BlogPost
```

Dans le dossier database/factories, un fichier apparaîtra avec le nom BlogPostFactory.php.

Dans la méthode de définition, nous modifierons le tableau de retour pour définir les données de notre article de blog, nous mettrons ce qui suit :

```

```

```
... use App\Models\User      ...
'title' => $this->faker->sentence, //Generates a fake sentence
'body' => $this->faker->paragraph(30), //generates fake 30 paragraphs
'user_id' => User::factory() //Generates a User from factory and extrac
...
```

Maintenant que nous avons créé l'usine, il est temps de créer un semoir (seed) pour ensemencer notre base de données. Nous allons le faire en utilisant PHP artisan bricoler.

Tinker est un outil de ligne de commande fourni avec Laravel pour permettre la manipulation de données sans modifier le code pendant le développement, c'est un bon outil pour faire des relations d'amorçage et de test.

Pour ouvrir Tinker, tapez :

```
php artisan tinker
```

Cela lancera une ligne de commande qui ressemble à ceci :

Commencez à taper votre code et appuyez sur Entrée pour l'exécuter.

```
\App\Models\BlogPost::factory()->times(10)->create();
```

ctrl+c pour sortir de Shell

Cela générera 10 articles de blog et les enregistrera dans la base de données et générera également 10 utilisateurs, chaque utilisateur possédera un article de blog.

---

7 - Création de contrôleurs

Les contrôleurs nous aident à effectuer des actions de manipulation de ressources, telles que CRUD Ops. Pour créer un contrôleur, nous utilisons la commande make:controller suivie du nom du contrôleur, **pour associer le contrôleur à un modèle, vous utilisez l'indicateur -m suivi du nom du modèle.**

```
php artisan make:controller BlogPostController -m BlogPost
```

关联的Model 名称

La convention de dénomination des contrôleurs dans Laravel est ModelName suivi du nom Controller. Pour le modèle BlogPost.php, le contrôleur sera BlogPostController.php.

Cela créera un fichier appelé BlogPostController.php dans le dossier app/Http/Controllers.

Le fichier ressemblera à ceci :

```
namespace App\Http\Controllers;
use App\Models\BlogPost;
use Illuminate\Http\Request;
class BlogPostController extends Controller
{

    public function index()
    {
        // afficher tous les articles du blog
    }
    public function create()
    {
        //afficher le formulaire pour créer un article de blog
    }

    public function store(Request $request)
    {
        //saisir un nouveau article
    }
    public function show(BlogPost $blogPost)
    {
        //afficher un article de blog
    }

    public function edit(BlogPost $blogPost)
    {
        //afficher le formulaire pour modifier l'article
    }

public function update(Request $request, BlogPost $blogPost)
    {
        //enregistrer l'article modifié
    }

    public function destroy(BlogPost $blogPost)
    {
        //supprimer un article
    }}
```

Le fichier sera créé avec toutes les méthodes de manipulation de ressources disponibles et le modèle BlogPost injecté dans le fichier par défaut.

---

8 - Travailler avec des routes

Maintenant que nous avons créé notre contrôleur, ciblons l'une des méthodes, disons index() en utilisant une route.

Toutes les routes Web sont stockées dans le fichier routes/web.php.

Ouvrez le fichier et vous verrez la route par défaut (racine) vers notre application, juste en dessous de la route racine, nous créerons la route du blog qui ouvrira le blog et affichera tous les articles disponibles.

Pour montrer que nous allons cibler la méthode index à l'intérieur de la classe BlogPostController.php.

Ce sera un itinéraire d'obtention puisque nous récupérons des données.

Notre fichier routes/web.php ressemblera à ceci :

```
Route::get('/blog', [\App\Http\Controllers\BlogPostController::class
, 'index']);
```

---

9 - Affichage de tous les articles de blog avec la méthode index()

```
...
public function index()
{
 $posts = BlogPost::all(); //récupérer tous les articles de blog de DB
 return $posts; //renvoie les messages récupérés
}
...
```

Si nous naviguons vers http://127.0.0.1:8000/blog, vous verrez un dump JSON des publications disponibles (10 publications).

ASTUCE : installez l'extension Chrome appelée JSON Formatter pour vous aider à formater le JSON vidé.

---

10 - Showing one blog post

Créez un itinéraire pour afficher 1 post.

Le parcours sera :

```
Route::get('/blog/{blogPost}', [\App\Http\Controllers\BlogPostControlle
```

Ici, nous avons introduit {blogPost}, c'est ce qu'on appelle un caractère générique. Cela signifie

que {blogPost} sera remplacé par tout ce qui est tapé après blog/ et cette valeur sera stockée dans une variable appelée $blogPost.

Sur la méthode show, nous aurons :

```
public function show(BlogPost $blogPost)
{
 return $blogPost; //renvoie les articles récupéré$
```

Si nous visitons http://127.0.0.1:8000/blog/5, il récupérera automatiquement le BlogPost avec l'ID de 5 et le stockera dans $blogPost en tant qu'instance de BlogPost Model.

C'est ce qu'on appelle la liaison de modèle de route dans Laravel. Vous fournissez un itinéraire avec un caractère générique qui est remplacé par la valeur fournie dans l'URL, puis Laravel utilise cette valeur pour essayer de trouver l'enregistrement associé à cette valeur, en particulier l'enregistrement avec cet ID.

S'il n'est pas trouvé, vous obtenez une erreur 404.

AVERTISSEMENT : la clé que nous utilisons sur le caractère générique doit être le même nom que le nom de la variable dans la méthode show pour que la liaison modèle-route se produise. Par exemple, **si le caractère générique de la route est {blogPost}, le nom de la variable sur la méthode show(BlogPost $blogPost) de la fonction publique doit être $blogPost.**

---

11 - Utilisation des vues et conception de l'interface utilisateur

Laravel utilise un moteur de modélisation appelé blade qui est injecté dans HTML et finit par être évalué en tant que HTML.

Nous allons comparer la syntaxe que nous utilisons sur blade à celle des vues internes PHP classiques :

Action	Blade Syntax	PHP Syntax
echo (escaped)	{{ $var }}	<?php echo htmlspecialchars($var) ?>
echo	{!! $var !!}	<?php echo $var ?>
if comparison	@if($var) // @else // @endif	<?php if($var) ?> // <?php else ?> // <?php endif ?>

foreach	@foreach($vars as $var)
	//
	@endforeach
	
	Or
	
	@forelse($vars as $var)
	//do when $vars != null
	@empty
	//do when $vars is empty
	@endforelse
		<?php foreach($vars as $var) ?>
		
		//
		
		<?php endfoeach ?>

switch	@switch($i)
	@case(1)
	
	// First case...
	
	@break
	
	@case(2)
	
	//Second case...
	
	@break
	
	@default
	
	//Default case...
	
	@endswitch
		N/A

while	@while (true) //iterations here @endwhile	N/A

https://laravel.com/docs/8.x/blade#blade-directives

> **ASTUCE :** La syntaxe PHP est toujours acceptée dans les vues Laravel mais comme vous l'avez vu, elle est maladroite. L'utilisation de la syntaxe blade est préférable.

Blade a plus de termes et de directives que nous devons comprendre :

View – un fichier HTML dans Laravel, tel que l'interface utilisateur.

Layout - Il s'agit du squelette de l'application, il définit les principaux éléments tels que les en-têtes et les pieds de page pour plus de cohérence et inclut également les principaux scripts et styles.

Component – Les composants sont des vues réutilisables, il peut s'agir d'un bouton par exemple.

Directives de Blade et leur signification :

Directive	Syntax	When to use
@extends	@extends('layouts.app')	Specifies which layout the child view will use.
@section	@section('content') Text to inject on app layout on main content @endsection	Specifies the content that will be injected to the main layout by the extending view. The first arg is the key.
@yield	@yield('content')	Brings the text which was specified on @section in the child view to the parent view (app layout).
@component	@section('components.button', ['name' => 'Button Name']) @endsection	This brings a component to the view, the first argument is the name of the component, and the second is an associative array containing all the data which is to be passed to the component.
@include	@include('dir.item')	This is used to bring some views inside another view e.g. bringing in a modal-dialog.
@csrf	<form method='POST'...> @csrf //other form fields here </form>	This extends to a form text input named _csrf with a token key from server to help protect against CSRF attacks. Only use it in forms.
@method('verb')	<form method='POST' id='edit' ...> @method('PUT') //other form fields here </form>	This is used in edit or delete forms to help provide extra http verbs since HTML forms supports POST and GET only. You can use it to help support DELETE verb PUT verb PATCH verb

12 - Concevoir la mise en page de notre application

Dans le dossier resource/views, créez un nouveau dossier et nommez-le layouts, puis créez un fichier dans le dossier et nommez-le app.blade.php.

Vous trouverez ci-dessous le code final de l'apparence du fichier :

```
<!DOCTYPE html>
<html lang="en"
>    <head
>        <meta charset="utf-8"
>        <meta name="viewport" content=
"width=device-width, initial-scale=1"
>
        <title>{{ config('app.name') }}</title
>
        <!-- Fonts -->
        <link href=
"https://fonts.googleapis.com/css2?family=Nunito:wght@400;600;700&display=sw
 rel="stylesheet"
>        <link href="
https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css
"  crossorigin="anonymous" integrity="
sha384-F3w7mX95PdgyTmZZMECAngseQB83DfGTowi0iMjiWaeVhAn4FJkqJByhZMI3AhiU
" rel="stylesheet" >

        <style
>            body
 {                font-family: 'Nunito'
;
            }        </style
>
    </head
>
    <body
>
    @yield('content')
    </body
>    <script src="
https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js
" integrity="
sha384-/bQdsTh/da6pkI1MST/rWKFNjaCP5gBSY4sEBT38Q/9RBh9AH40zEOg7Hlq2THRZ
"  crossorigin="anonymous"></script>
</html>
```

Avec cela, nous avons créé notre mise en page, c'est juste une page HTML avec la police Google et Bootstrap dedans.

Au titre {{ config('app.name') }} – Est une méthode d'aide à l'accesseur de configuration Laravel pour accéder au nom de notre application qui apparaît comme le titre de notre page.

Dans le corps, @yield('content') - est une directive de blade qui sera utilisée pour amener le contenu des vues enfants vers la mise en page.

En savoir plus sur le bootstrap sur: https://getbootstrap.com/docs/5.1/getting-started/introduction/

---

13 - Conception de la page d'accueil

C'est la page que nous avons vue lors de la création de notre première application, nous allons la

reconcevoir pour afficher la page d'accueil. Nous allons étendre notre mise en page en utilisant la directive @extend. Il est situé dans le dossier resources/views, nommé welcome.blade.php.

Le code final de la page ressemblera à ceci :

```
@extends('layouts.app')
@section('content')    <div class="container"
>        <div class="row"
>            <div class="col-12 text-center pt-5"
>                <h1 class="display-one mt-5"
>{{ config('app.name') }}</h1
>                <p>
Ce blog génial contient de nombreux articles, cliquez sur le bouton ci-
</p
>                <br
>                <a href="/blog" class="btn btn-outline-primary">
Afficher le blog</a
>            </div
>        </div
>    </div
>@endsection
```

---

14 - Designing the blog page

In this section, we will return a view to the user, to do this, we can use a method that Laravel gives. Instaed of saying return $posts we say return view ('view.name', [$data]); so, we will modify the code in BlogPostController.php within the index() method to return a view instead of a json file.

Before we modify the code, first go to resources/views folder and create a folder named blog and in that folder create a view file named index.blade.php, this will be our index method view or the view to show all the blog posts .

Then modify the code in your BlogPostController.php in index() method to look like this:

```
public
 function index()
{
 $posts = BlogPost::all(); //fetch all blog posts from DBreturn view(
'blog.index', [
            'posts' => $posts,
        ]); //renvoie la vue avec les articles}
```

Avec cela, nous aurons accès à une variable appelée $posts dans notre vue qui est un objet contenant plusieurs articles de blog.

Conception de la page des articles de blog.

```
@extends('layouts.app')
@section('content')    <div class="container"
>        <div class="row"
>            <div class="col-12 pt-2"
>                <div class="row"
>                    <div class="col-8"
>                        <h1 class="display-one">Notre blog!</h1
>                        <p>Bonne lecture de nos articles.
Cliquez sur un article pour lire!</p
>                    </div
>                    <div class="col-4"
>                        <p>Créer un nouveau message</p
>                        <a href="/blog/create/post" class=
"btn btn-primary btn-sm">Ajouter un message </a
>                    </div
>                </div
>
                @forelse($posts as $post)                    <ul
>                        <li><a href="./blog/{{ $post->id }}"
>{{ ucfirst($post->title) }}</a></li
>                    </ul
>
                @empty                    <p class="text-warning">
Aucun article de blog disponible </p
>
                @endforelse            </div
>        </div
>    </div
>@endsection
```

Dans cette page, la lame parcourra les publications (si elle n'est pas nulle) et crachera un lien vers la publication en particulier et avec un titre de publication comme texte du lien.

Il attachera l'ID de la publication au lien.

- ./blog/{{ $post->id }} – Un identifiant BlogPost est attaché aux URL de sorte que l'exemple d'URL formé pour le post 5 sera http://127.0.0.1:8000/blog/5
- {{ ucfirst($post->title) }} – Titre du message formaté avec chaque première lettre en majuscule.
- Le bouton Add Post nous aidera à créer une nouvelle publication.

---

15 - Conception de la page BlogPost

Dans notre page d'article de blog actuelle, nous renvoyons toujours des données json brutes à l'utilisateur, dans cette section, nous renverrons une vue à l'utilisateur.

Nous allons modifier le code dans BlogPostController.php dans la méthode show() pour renvoyer une vue au lieu des données json.

Avant de modifier le code, nous devons d'abord aller dans le dossier resources/views/blog pour créer un fichier de vue nommé show.blade.php, ce sera notre vue de méthode d'affichage ou la vue à afficher dans un article de blog particulier.

Modifiez ensuite le code dans votre BlogPostController.php dans la méthode show() pour ressembler à ceci :

```
public function show(BlogPost $blogPost)
{
    return view('blog.show', [
        'post' => $blogPost,
    ]); //renvoie la vue avec l'article}
```

Avec cela, nous aurons accès à une variable appelée $post dans notre vue qui est l'objet contenant le article du blog que nous voulons afficher.

Concevons maintenant notre page d'article de blog.

Le code ressemblera à ceci :

```
@extends('layouts.app')
@section('content')    <div class="container"
>        <div class="row"
>            <div class="col-12 pt-2"
>                <a href="/blog" class=
"btn btn-outline-primary btn-sm">Retourner</a
>                <h1 class="display-one"
>{{ ucfirst($post->title) }}</h1
>                <p>{!! $post->body !!}</p
>                 <hr
>                <a href="/blog/{{ $post->id }}/edit" class=
"btn btn-outline-primary">Modifier la publication</a
>                <br><br
>                <form id="delete-frm" class="" action="" method=
"POST"
>
                    @method('DELETE')
                    @csrf                    <button class=
"btn btn-danger">Supprimer le message</button
>                </form
>            </div
>        </div
>    </div
>@endsection
```

- {!! $post->body !!} – Nous avons utilisé cette directive pour nous assurer que nous permettons au HTML dans le corps d'être affiché en gras.
- id }}/edit" class="btn btn-outline-primary">Modifier le message - Ce bouton sera cliqué pour modifier le message.
- Formulaire de suppression : Ce formulaire sera utilisé pour supprimer la publication. La directive @method('DELETE') crée un champ qui remplacera la méthode de publication par défaut par la méthode DELETE. Il en sera de même pour la directive @csrf.

- Le bouton Go Back nous ramènera à la page Blog.

---

16 - Travailler avec d'autres méthodes

Jusqu'à présent, nous n'avons travaillé qu'avec des routes get. Les méthodes create() et edit() sont censées afficher les formulaires de création et d'édition de manière respectueuse.

La méthode store() sera un verbe post http puisque nous publierons le formulaire create BlogPost pour stocker les données, la méthode update() aura besoin d'un verbe put ou patch pour mettre à jour les données et la méthode destroy() nécessitera une suppression verbe pour supprimer le message.

> **ASTUCE :** les verbes HTTP sont également appelés méthodes ou actions, ils sont normalement utilisés pour définir l'action qui est entreprise sur le serveur.
>
> Par exemple. Un verbe/action/méthode POST sera utilisé pour publier des données sur le serveur, une méthode GET sera utilisée pour obtenir les données du serveur, la méthode PATCH/PUT sera utilisée pour mettre à jour les données et la méthode DELETE sera utilisée pour supprimer les données du serveur .
>
> Il existe d'autres verbes, mais vous les utiliserez rarement, même dans un environnement de développement professionnel.

Nous les implémenterons après avoir créé l'interface utilisateur, pour le moment, nous pouvons créer leurs itinéraires pour les utiliser plus tard.

```
Route::get('/blog/create/post', [\App\Http\Controllers\BlogPostControll
Route::post('/blog/create/post', [\App\Http\Controllers\BlogPostControl
Route::get('/blog/{blogPost}/edit', [\App\Http\Controllers\BlogPostCont
Route::put('/blog/{blogPost}/edit', [\App\Http\Controllers\BlogPostCont
Route::delete('/blog/{blogPost}', [\App\Http\Controllers\BlogPostContro
```

---

17 - Créer une nouvelle page de publication

Nous avons déjà créé une route pour cette page http://127.0.0.1:8000/blog/create/post.

Nous allons d'abord modifier le code dans le fichier BlogPostController.php dans la méthode create() pour renvoyer la vue.

Avant de modifier le code, allez d'abord dans le dossier resources/views/blog et créez un fichier de vue nommé create.blade.php, ce sera notre vue de méthode de création ou la vue pour afficher un formulaire nécessaire pour créer un article de blog.

Le code de la méthode create() ressemblera à ceci :

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

Concevons maintenant notre vue.

Le code ressemblera à ceci :

```
@extends('layouts.app')
@section('content')
    <div class="container"
>         <div class="row"
>             <div class="col-12 pt-2"
>                 <a href="/blog" class="btn btn-outline-primary btn-sm">
Retourner</a
>                 <div class="border rounded mt-5 pl-4 pr-4 pt-4 pb-4"
>                     <h1 class="display-4">Créer un nouveau message </h1
>                     <p>
Remplissez et soumettez ce formulaire pour créer un article </p
>
                    <hr
>
                    <form action="" method="POST"
>
                        @csrf                        <div class="row"
>                             <div class="control-group col-12"
>                                 <label for="title">Titre du message</
label
>                                 <input type="text" id="title" class=
"form-control" name="title"
                                        placeholder="
Entrer le titre du message" required
>                                 </div
>                                 <div class="control-group col-12 mt-2"
>                                     <label for="body">Corps du message</
label
>                                     <textarea id="body" class="form-control"
 name="body" placeholder="Entrer le corps du message"
                                               rows="" required></textarea
>                                 </div
>                             </div
>                             <div class="row mt-2"
>                                 <div class=
"control-group col-12 text-center"
>                                     <button id="btn-submit" class=
"btn btn-primary"
>                                         Créer un message
                                    </button
>                                 </div
>                             </div
>                     </form
>                 </div
>
            </div
>         </div
>     </div
>
```

```
@endsection
```

Ce formulaire soumettra une requête POST à cette route http://127.0.0.1:8000/blog/create/post.

La directive @csrf se développera dans le navigateur pour nous donner le champ de jeton dans le formulaire.

---

18 - Accepter et enregistrer le message soumis

Dans notre BlogPostController.php dans la méthode store(), nous implémenterons le code pour enregistrer la publication dans la base de données et rediriger l'utilisateur vers la publication créée.

Le code ressemblera à ceci :

```
...
public function store(Request $request)
    {
        $newPost = BlogPost::create([
            'title' => $request->title,
            'body' => $request->body,
            'user_id' => 1
        ]);
        return redirect('blog/' . $newPost->id);
    }...
```

Ici, nous utilisons la méthode statique Model::create() qui accepte un tableau associatif avec les clés étant le champ de la table et la valeur étant les données à insérer dans la table.

Ici, nous attribuons notre publication à user_id 1. Vous pouvez en savoir plus sur l'authentification Laravel plus tard pour savoir comment associer une publication à l'utilisateur connecté, Laravel dispose de nombreuses techniques d'authentification.

La valeur de retour est une redirection qui redirigera vers notre itinéraire de publication unique avec l'ID de la publication. Maintenant, avant de terminer, nous devons modifier notre modèle (BlogPost.php) pour afficher les champs remplissables afin de les protéger des entrées indésirables.

Le modèle modifié ressemblera à ceci :

```
<?php
namespace App\Models
;
use Illuminate\Database\Eloquent\Factories\HasFactory
;use Illuminate\Database\Eloquent\Model
;
class BlogPost extends Model
{    use HasFactory
;
    protected $fillable = ['title', 'body', 'user_id'
];}
```

19 - Modification d'un message

Nous avons déjà créé une route pour cette page http://127.0.0.1:8000/blog/{blogPost}/edit.

Nous allons d'abord modifier le code dans le fichier BlogPostController.php dans la méthode edit() pour retourner la vue.

Avant de modifier le code, allez d'abord dans le dossier resources/views/blog, créez un fichier de vue nommé edit.blade.php, ce sera notre vue de méthode d'édition ou la vue pour afficher un formulaire pour éditer un article de blog.

Modifiez ensuite le code dans votre fichier BlogPostController.php dans la méthode edit() pour ressembler à ceci :

```
...
public function edit(BlogPost $blogPost)
{
 return view(‘blog.edit’, [
‘post’ => $blogPost,]); //renvoie la vue d'édition avec la publication

}
...
```

Avec cela, nous aurons accès à une variable appelée $post dans notre vue qui est l'objet contenant le billet de blog que nous voulons éditer.

La vue ressemblera à ceci :

```
@extends('layouts.app')
@section('content')
    <div class="container"
>         <div class="row"
>             <div class="col-12 pt-2"
>                 <a href="/blog" class="btn btn-outline-primary btn-sm">
Retour</a
>                 <div class="border rounded mt-5 pl-4 pr-4 pt-4 pb-4"
>                     <h1 class="display-4">Modifier la publication </h1
>                     <p>
Modifier et soumettre ce formulaire pour mettre à jour un message </p
>
                    <hr
>
                    <form action="" method="POST"
>
                        @csrf
                        @method('PUT')                            <div class=
"row"
>                             <div class="control-group col-12"
>                                 <label for="title">Titre du message</
label
>                                 <input type="text" id="title" class=
"form-control" name="title"
                                        placeholder="
Entrer le titre du message " value="{{ $post->title }}" required
>
```

```
                            </div
>                            <div class="control-group col-12 mt-2"
>                                <label for="body">Corps du message </
label
>                                <textarea id="body" class="form-control"
 name="body" placeholder="Entrer le corps du message"
                                          rows="5" required
>{{ $post->body }}</textarea
>                            </div
>                        </div
>                        <div class="row mt-2"
>                            <div class=
"control-group col-12 text-center"
>                                <button id="btn-submit" class=
"btn btn-primary"
>                                    Mettre à jour le message
                                </button
>                            </div
>                        </div
>                    </form
>                </div
>
            </div
>        </div
>    </div
>
@endsection
```

Cela affichera un formulaire pré-rempli.

@method('PUT') - Cela s'étendra à un champ de saisie qui sera utilisé pour remplacer le verbe POST par défaut comme nous l'avons vu avec @method('DELETE').

---

20 - Mise à jour du message

Dans notre fichier BlogPostController.php dans notre méthode update(), nous allons implémenter le code pour enregistrer le message dans la base de données, puis rediriger l'utilisateur vers le message édité.

Le code ressemblera à ceci :

```
...
public function update(Request $request, BlogPost $blogPost)
    {
        $blogPost->update([
            'title' => $request->title,
            'body' => $request->body
        ]);
        return redirect('blog/' . $blogPost->id);
    }
...
```

Ici, nous utilisons la méthode $modelInstance->update() qui accepte un tableau associatif avec les clés du champ de la table et la valeur sera les données que nous mettons à jour.

---

21 - Supprimer un message

Dans notre fichier BlogPostController.php dans notre méthode destroy(), nous allons implémenter le code pour enregistrer le message dans la base de données, puis rediriger l'utilisateur vers le message édité.

Le code ressemblera à ceci :

```
...
public function destroy(BlogPost $blogPost)
    {
        $blogPost->delete();
        return redirect('/blog');
    }
...
```

Ici, nous utilisons la méthode $modelInstance->delete() qui supprimera le message de la base de données.

Référence: https://www.section.io/engineering-education/laravel-beginners-guide-blogpost/
