**TP : Création d'une Application de Blog avec Laravel 10**

Pour créer une application de blog complète avec Laravel 10, nous allons détailler chaque étape du TP, y compris le code nécessaire pour les fichiers de migration, et expliquer le rôle des commandes utilisées pour l'authentification et le front-end.

**1. Installation et Configuration Initiale**

Après avoir créé un nouveau projet Laravel avec **composer create-project --prefer-dist laravel/laravel blog**, vous devez configurer votre environnement de base de données dans le fichier **.env** à la racine de votre projet. Voici un exemple de configuration pour une base de données MySQL :

DB\_CONNECTION=mysql DB\_HOST=127.0.0.1 DB\_PORT=3306 DB\_DATABASE=nom\_de\_la\_base\_de\_donnees DB\_USERNAME=utilisateur\_de\_la\_base DB\_PASSWORD=mot\_de\_passe

**2. Modèles et Migrations**

Création de modèles et migrations

Pour créer un modèle **Article** avec sa migration :

php artisan make:model Article -m

Pour le modèle **Commentaire** :

php artisan make:model Commentaire -m

Contenu des fichiers de migration

Pour **articles** :

    public function up()

    {

        Schema::create('articles', function (Blueprint $table) {

            $table->id();

            $table->string('titre');

            $table->text('contenu');

            $table->foreignId('user\_id')->constrained()->onDelete('cascade');

            $table->timestamps();

        });

    }

Pour **commentaires** :

    public function up()

    {

        Schema::create('commentaires', function (Blueprint $table) {

            $table->id();

            $table->text('contenu');

            $table->foreignId('article\_id')->constrained()->onDelete('cascade');

            $table->foreignId('user\_id')->constrained()->onDelete('cascade');

            $table->timestamps();

        });

    }

Exécutez les migrations avec **php artisan migrate**.

Création des modèles :

class Commentaire extends Model

{

    use HasFactory;

    // Protection des attributs pour l'assignation massive

    protected $fillable = ['contenu', 'article\_id', 'user\_id'];

    /\*\*

     \* Relation "Un commentaire appartient à un article"

     \*/

    public function article()

    {

        return $this->belongsTo(Article::class);

    }

    /\*\*

     \* Relation "Un commentaire appartient à un utilisateur"

     \*/

    public function user()

    {

        return $this->belongsTo(User::class);

    }

}

Modèle commentaire :

class Article extends Model

{

    use HasFactory;

    // Protection des attributs pour l'assignation massive

    protected $fillable = ['titre', 'contenu', 'user\_id'];

    /\*\*

     \* Relation "Un article appartient à un utilisateur"

     \*/

    public function user()

    {

        return $this->belongsTo(User::class);

    }

    /\*\*

     \* Relation "Un article a plusieurs commentaires"

     \*/

    public function commentaires()

    {

        return $this->hasMany(Commentaire::class);

    }

}

**3. Authentification avec Laravel Breeze**

Installation de Laravel Breeze

composer require laravel/breeze --dev

php artisan breeze:install

Ces commandes installent Laravel Breeze et publient les vues d'authentification, les routes, et les contrôleurs nécessaires. Breeze est une solution d'authentification simple proposée par Laravel.

Configuration du Front-End

npm install && npm run dev

Ces commandes installent les dépendances NPM et compilent vos assets. Breeze utilise TailwindCSS par défaut, donc cette étape prépare également le style de votre application.

**4. Développement des Fonctionnalités du Blog**

Routes

Dans **routes/web.php**, définissez les routes nécessaires pour votre blog :

Route::get('/', [ArticleController::class, 'index'])->name('articles.index');

Route::middleware(['auth'])->group(function () {

Route::get('/articles/create', [ArticleController::class, 'create'])->name('articles.create'); Route::post('/articles', [ArticleController::class, 'store'])->name('articles.store'); Route::get('/articles/{article}', [ArticleController::class, 'show'])->name('articles.show'); });

Contrôleurs

Créez des contrôleurs pour gérer la logique de votre application :

php artisan make:controller ArticleController

php artisan make:controller CommentaireController

Dans **ArticleController**, par exemple, vous aurez des méthodes pour **index**, **create**, **store**, et **show**.

Par exemple:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Models\Article;

class ArticleController extends Controller

{

    public function index()

    {

        // Récupération de tous les articles avec pagination

        $articles = Article::latest()->paginate(10);

        return view('articles.index', compact('articles'));

    }

    public function create()

    {

        // Affichage du formulaire de création d'article

        return view('articles.create');

    }

    public function store(Request $request)

    {

        // Validation des données du formulaire

        $request->validate([

            'titre' => 'required|string|max:255',

            'contenu' => 'required',

        ]);

        // Création d'un nouvel article

        $article = new Article();

        $article->titre = $request->titre;

        $article->contenu = $request->contenu;

        $article->user\_id = auth()->id(); // Supposant que l'utilisateur doit être connecté

        $article->save();

        return redirect()->route('articles.index');

    }

    public function show(Article $article)

    {

        // Affichage d'un article spécifique et ses commentaires

        return view('articles.show', compact('article'));

    }

    public function edit(Article $article)

    {

        // Affichage du formulaire d'édition d'article

        return view('articles.edit', compact('article'));

    }

    public function update(Request $request, Article $article)

    {

        // Validation des données du formulaire

        $request->validate([

            'titre' => 'required|string|max:255',

            'contenu' => 'required',

        ]);

        // Mise à jour de l'article

        $article->update($request->all());

        return redirect()->route('articles.index')->with('success', 'Article mis à jour avec succès.');

    }

    public function destroy(Article $article)

    {

        // Suppression de l'article

        $article->delete();

        return back()->with('success', 'Article supprimé avec succès.');

    }

}

CommentaireController

<?php

namespace App\Http\Controllers;

use App\Models\Article;

use App\Models\Commentaire;

use Illuminate\Http\Request;

class CommentaireController extends Controller

{

    public function store(Request $request, Article $article)

    {

        // Validation des données du formulaire

        $request->validate([

            'contenu' => 'required',

        ]);

        // Création d'un nouveau commentaire pour l'article

        $commentaire = new Commentaire();

        $commentaire->contenu = $request->contenu;

        $commentaire->user\_id = auth()->id(); // Supposant que l'utilisateur doit être connecté

        $commentaire->article\_id = $article->id;

        $commentaire->save();

        return redirect()->route('articles.show', $article)->with('success', 'Commentaire ajouté avec succès.');

    }

}

Vues

Utilisez Blade pour créer vos vues dans **resources/views/articles**. Vous pouvez créer des fichiers tels que **index.blade.php**, **create.blade.php**, et **show.blade.php** pour afficher, créer, et voir les articles, respectivement.

Pour index.blade.php, affichez tous les articles :

@extends('layouts.app')

@section('content')

<div class="container">

    <h1>Articles</h1>

    <a href="{{ route('articles.create') }}" class="btn btn-primary">Créer un nouvel article</a>

    @foreach ($articles as $article)

        <div class="article">

            <h2><a href="{{ route('articles.show', $article) }}">{{ $article->titre }}</a></h2>

            <p>{{ $article->contenu }}</p>

        </div>

    @endforeach

    {{ $articles->links() }}

</div>

@endsection

**Create.blade.php**

@extends('layouts.app')

@section('content')

<div class="container">

    <h1>Créer un nouvel article</h1>

    <form method="POST" action="{{ route('articles.store') }}">

        @csrf

        <div class="form-group">

            <label for="titre">Titre</label>

            <input type="text" class="form-control" id="titre" name="titre" required>

        </div>

        <div class="form-group">

            <label for="contenu">Contenu</label>

            <textarea class="form-control" id="contenu" name="contenu" rows="5" required></textarea>

        </div>

        <button type="submit" class="btn btn-primary">Publier</button>

    </form>

</div>

@endsection

**Show.blade.php**

@extends('layouts.app')

@section('content')

<div class="container">

    <h1>{{ $article->titre }}</h1>

    <p>{{ $article->contenu }}</p>

    <h3>Commentaires</h3>

    @foreach ($article->commentaires as $commentaire)

        <div class="commentaire">

            <p>{{ $commentaire->contenu }}</p>

            <p>Par {{ $commentaire->user->name }}, le {{ $commentaire->created\_at->format('d/m/Y') }}</p>

        </div>

    @endforeach

    @auth

    <form method="POST" action="{{ route('commentaires.store', $article) }}">

        @csrf

        <div class="form-group">

            <label for="contenu">Ajouter un commentaire</label>

            <textarea class="form-control" id="contenu" name="contenu" rows="3" required></textarea>

        </div>

        <button type="submit" class="btn btn-primary">Commenter</button>

    </form>

    @endauth

</div>

@endsection

**5. Gestion des Commentaires**

Assurez-vous que votre **CommentaireController** contient la logique pour stocker les commentaires, et modifiez **show.blade.php** pour inclure un formulaire de commentaire et afficher les commentaires existants.

**6. Tests**

Pour tester la création d'un article, vous pouvez utiliser PHPUnit en créant un test avec :

php artisan make:test ArticleTest

Dans **tests/Feature/ArticleTest.php**, écrivez un test pour simuler la création d'un article et vérifiez que l'article est bien créé.

public function test\_article\_creation()

{

$response = $this->post('/articles', [

'titre' => 'Un nouveau titre',

'contenu' => 'Du contenu d\'article',

'user\_id' => 1, // Assurez-vous que cet utilisateur existe ou utilisez une factory

]);

$response->assertRedirect('/chemin-vers-quelque-part');

$this->assertDatabaseHas('articles', ['titre' => 'Un nouveau titre']);

}

**Partie II**

1 - Dans routes/web.php, ajoutez les routes pour l'édition et la suppression :

Route::get('/articles/{article}/edit', [ArticleController::class, 'edit'])->name('articles.edit');

Route::put('/articles/{article}', [ArticleController::class, 'update'])->name('articles.update');

Route::delete('/articles/{article}', [ArticleController::class, 'destroy'])->name('articles.destroy');

2- Dans ArticleController, ajoutez les méthodes edit, update, et destroy pour gérer ces actions.

public function edit(Article $article)

{ return view('articles.edit', compact('article')); }

public function update(Request $request, Article $article)

{ $article->update($request->all());

return redirect()->route('articles.index'); }

3 - Et pour destroy :

public function destroy(Article $article)

{ $article->delete();

return redirect()->route('articles.index'); }

4 - Création de la Vue d'Édition

Créez une vue edit.blade.php dans resources/views/articles pour le formulaire d'édition d'article, similaire à votre formulaire de création mais pré-rempli avec les données existantes de l'article.

5 - Dans ArticleController au niveau de la méthode index, modifiez la récupération des articles pour implémenter la pagination :

public function index()

{

$articles = Article::paginate(10); // 10 articles par page

return view('articles.index', compact('articles'));

}

Dans la vue, utilisez le composant de pagination de Blade pour afficher les liens de pagination :

{{ $articles->links() }}

6 - Tests de Fonctionnalité pour l'Édition et la Suppression

Créez des tests pour vérifier que les fonctionnalités d'édition et de suppression fonctionnent comme attendu. Par exemple, pour tester la suppression :

/\*\* @test \*/

public function a\_user\_can\_delete\_an\_article()

{

$article = Article::factory()->create();

$this->delete('/articles/' . $article->id)

->assertRedirect('/route-après-suppression');

$this->assertDatabaseMissing('articles', ['id' => $article->id]);

}