Étape 1 : Préparation de la Base de Données

1. **Migrations :** Créez des migrations pour les tables **articles** et **comments** :

php artisan make:migration create_articles_table

php artisan make:migration create_comments_table

```
Schema::create('articles', function (Blueprint $table) {
   $table->id();
   $table->string('title');
   $table->text('body');
   $table->timestamps(); // Crée les champs created_at et updated_at
});
Schema::create('comments', function (Blueprint $table) {
   $table->id();
   $table->foreignId('article_id')->constrained()->onDelete('cascade');
   $table->text('content');
   $table->timestamps();
});
```

Notez l'utilisation de foreignId('article_id')->constrained() pour définir une clé étrangère qui référence la table articles. onDelete('cascade') garantit la suppression des commentaires associés si un article est supprimé.

2. **Modèles :** Créez des modèles pour **Article** et **Comment** :

php artisan make:model Article

php artisan make:model Comment

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
   public function comments()
   {
      return $this->hasMany(Comment::class);
   }
}
```

```
namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
   public function article()
   {
      return $this->belongsTo(Article::class);
   }
}
```

Étape 2 : Création des Contrôleurs

1. **Contrôleurs :** Générez des contrôleurs pour gérer la logique des articles et des commentaires :

php artisan make:controller ArticleController --resource

php artisan make:controller CommentController

Étape 3 : Implémentation des Fonctionnalités

1. **CRUD Articles :** Dans **ArticleController**, implémentez les méthodes pour créer, lire, mettre à jour, et supprimer des articles. Utilisez des formulaires pour la création et la mise à jour.
2. **Commentaires :** Dans **CommentController**, ajoutez une méthode pour poster des commentaires sous les articles. Assurez-vous que seuls les utilisateurs authentifiés (éditeurs et visiteurs) peuvent commenter.

Étape 4 : Application des Middlewares

L'objectif est d'appliquer les règles suivantes :

- **Admins** peuvent créer, éditer, et supprimer des articles.
- **Éditeurs** peuvent éditer des articles et commenter.
- **Visiteurs** peuvent uniquement voir les articles et commenter.

Création des Middlewares : Générez trois middlewares pour chaque type d'utilisateur :

php artisan make:middleware AdminMiddleware

php artisan make:middleware EditorMiddleware

php artisan make:middleware VisitorMiddleware

php artisan make:middleware EditorMiddleware

Étape 1 : Configuration des Routes

Ouvrez le fichier routes/web.php et configurez les routes en appliquant les middlewares appropriés :

```
// Route accessible par tous les utilisateurs
Route::get('/articles', 'ArticleController@index');

// Routes protégées par le middleware 'admin'
Route::group(['middleware' => ['admin']], function () {
    Route::get('/article/create', 'ArticleController@create');
    Route::post('/articles', 'ArticleController@store');
    Route::get('/article/edit/{id}', 'ArticleController@edit');
    Route::put('/article/update/{id}', 'ArticleController@update');
    Route::delete('/article/delete/{id}', 'ArticleController@destroy');
});

// Routes protégées par le middleware 'editor', permettant de commenter et éditer des articles
Route::group(['middleware' => ['editor']], function () {
    Route::post('/comment/store', 'CommentController@store');
    // Supposant que les éditeurs peuvent également éditer certains articles
    Route::get('/article/edit/{id}', 'ArticleController@edit')->middleware('canEditArticle');
    Route::put('/article/update/{id}', 'ArticleController@update')->middleware('canEditArticle');
});

// Route pour commenter accessible par les éditeurs et visiteurs
Route::post('/comment/store', 'CommentController@store')->middleware(['auth']);
```

Étape 2 : Configuration des Middlewares

Dans chaque middleware (AdminMiddleware, EditorMiddleware), définissez la logique pour vérifier le rôle de l'utilisateur. Vous avez déjà un exemple de middleware pour les admins. Voici un exemple pour le middleware EditorMiddleware :

```
public function handle($request, Closure $next)
{
    if (auth()->user() && auth()->user()->role == 'editor') {
```

```
        return $next($request);
    }
    return redirect('home')->with('error', 'Accès non autorisé.');
}
```

Pour le middleware **CanEditArticleMiddleware**

```
public function handle($request, Closure $next)
{
    $article = \App\Models\Article::find($request->route('article'));
    if ($article && (auth()->user()->id === $article->user_id || auth()->user()->role === 'editor')) {
        return $next($request);
    }
    return redirect('articles')->with('error', 'Vous n\'avez pas la permission d\'éditer cet article.');
}
```

Étape 3 : Enregistrement des Middlewares

Enregistrez vos middlewares dans **app/Http/Kernel.php** sous le tableau **$routeMiddleware** si ce n'est pas déjà fait :

```
protected $routeMiddleware = [
    // autres middlewares...
    'admin' => \App\Http\Middleware\AdminMiddleware::class,
    'editor' => \App\Http\Middleware\EditorMiddleware::class,
    'visitor' => \App\Http\Middleware\VisitorMiddleware::class,
    'canEditArticle' => \App\Http\Middleware\CanEditArticleMiddleware::class,
];
```

Étape 4 : Contrôleurs

Dans vos contrôleurs (ArticleController et CommentController), assurez-vous de gérer correctement les actions en fonction des permissions. Par exemple, dans le ArticleController@store, vérifiez si l'utilisateur a la permission de créer un article :

```
public function store(Request $request)
{
    // Validation et logique de stockage de l'article
    if (auth()->user()->role !== 'admin') {
        return redirect('articles')->with('error', 'Action non autorisée.');
    }
```

```
    // Création de l'article...

    return redirect('articles')->with('success', 'Article créé avec succès.');

}
```

Liste des articles avec commentaires :

En utilisant **Article::with('comments')->get();**, chaque article récupéré contiendra également ses commentaires associés, ce qui permet un accès direct aux commentaires dans la vue.

```
public function index()
    {
        // Récupérer tous les articles avec leurs commentaires
        $articles = Article::with('comments')->get();

        // Passer les articles à la vue
        return view('articles.index', compact('articles'));
    }
```

Étape 5 : Routes et Vues

1. **Routes :** Définissez des routes pour les articles et les commentaires dans **routes/web.php**. Assurez-vous d'utiliser les middlewares pour protéger ces routes.
2. **Vues :** Créez des vues pour afficher les articles, les formulaires de création et d'édition, et pour montrer les commentaires sous chaque article.

Création de la vue **resources/views/articles/index.blade.php** pour afficher les commentaires sous chaque article :

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Liste des Articles</title>

</head>

<body>

    <h1>Liste des Articles</h1>

    @foreach($articles as $article)

        <div>

            <h2>{{ $article->title }}</h2>

            <p>{{ $article->body }}</p>
```

```
        <h3>Commentaires</h3>
        <ul>
          @forelse($article->comments as $comment)
            <li>{{ $comment->content }} –
                <small>
                    {{ $comment->created_at->format('d/m/Y H:i') }}
                </small>
            </li>
          @empty
            <li>Pas de commentaires pour cet article.</li>
          @endforelse
        </ul>
      </div>
  @endforeach
</body>
</html>
```

Étape 6 : Pris en charge des images aussi des articles

1. Création d'une Nouvelle Migration pour Ajouter la Colonne

Utilisez la commande Artisan pour créer une migration qui ajoutera la colonne image à la table articles :

php artisan make:migration add_image_to_articles_table --table=articles

2. Modification de la Migration

```
public function up()
  {
    Schema::table('articles', function (Blueprint $table) {
      $table->string('image')->nullable()->after('body');
      // Ajoute une colonne 'image' après la colonne 'body'
    });
  }

  public function down()
  {
    Schema::table('articles', function (Blueprint $table) {
      $table->dropColumn('image');
    });
  }
```

Exécution de la migration :
php artisan migrate

3.Modification de la Méthode store du Contrôleur pour Enregistrer l'URL de l'Image
use Illuminate\Http\Request;

use App\Models\Article;

```
public function store(Request $request)
{
   $request->validate([
      'title' => 'required',
      'body' => 'required',
      'image' => 'image|nullable|max:1999', // Valide l'image si elle est présente
   ]);

   // Gérer l'upload de l'image
   if ($request->hasFile('image')) {
      // Obtenir le nom de fichier avec l'extension
      $filenameWithExt = $request->file('image')->getClientOriginalName();
      // Obtenir juste le nom de fichier
      $filename = pathinfo($filenameWithExt, PATHINFO_FILENAME);
      // Obtenir juste l'extension
      $extension = $request->file('image')->getClientOriginalExtension();
      // Nom de fichier à stocker
      $fileNameToStore= $filename.'_'.time().'.'.$extension;
      // Upload de l'image
      $path = $request->file('image')->storeAs('public/images', $fileNameToStore);
   } else {
      $fileNameToStore = 'noimage.jpg';
   }

   // Créer l'article
   $article = new Article;
   $article->title = $request->input('title');
   $article->body = $request->input('body');
   $article->image = $fileNameToStore; // Sauvegarde de l'URL de l'image
   $article->save();

   return redirect('/articles')->with('success', 'Article créé avec succès.');
}
```

4.Modification de la Vue pour Afficher l'Image

```
    @if($article->image != 'noimage.jpg')
        <img src="/storage/images/{{ $article->image }}" alt="Image de l'article"
style="width:100px;height:100px;">
```

@endif

On utilise le système de fichiers par défaut de Laravel pour stocker les images dans **storage/app/public/images** (créer un lien symbolique vers public/storage en utilisant la commande **php artisan storage:link)**

Étape 7 : utilisation des factories et des seeders

Créer et utiliser des factories et des seeders pour insérer des données aléatoires dans les tables articles et comments.

1/

php artisan make:factory ArticleFactory --model=Article

Dans le fichier **database/factories/ArticleFactory.php**, définissez la structure des données aléatoires pour les articles :

```
use Faker\Generator as Faker;


$factory->define(App\Models\Article::class, function (Faker $faker) {
   return [
      'title' => $faker->sentence,
      'body' => $faker->paragraph,
   ];
});
```

De même pour la table comments

php artisan make:factory CommentFactory --model=Comment

```
use Faker\Generator as Faker;


$factory->define(App\Models\Comment::class, function (Faker $faker) {
   return [
      'article_id' => \App\Models\Article::inRandomOrder()->first()->id,
      'content' => $faker->text,
   ];
});
```

2/ **Générer les Seeders pour les Articles et les Commentaires** :

php artisan make:seeder ArticlesTableSeeder

php artisan make:seeder CommentsTableSeeder

Configurer le Seeder d'Articles dans **database/seeders/ArticlesTableSeeder.php** :

```
use Illuminate\Database\Seeder;


class ArticlesTableSeeder extends Seeder
{
   public function run()
   {
      factory(App\Models\Article::class, 50)->create(); // Génère 50 articles
   }
}
```

De même pour les commentaires

```
use Illuminate\Database\Seeder;


class CommentsTableSeeder extends Seeder
{
   public function run()
   {
      factory(App\Models\Comment::class, 200)->create(); // Génère 200 commentaires
   }
}
```

3/ **Enregistrer les Seeders dans DatabaseSeeder** :

Ouvrez le fichier **database/seeders/DatabaseSeeder.php** et assurez-vous d'appeler vos seeders dans la méthode **run** :

```
public function run()
{
   $this->call(ArticlesTableSeeder::class);
   $this->call(CommentsTableSeeder::class);
}
```

4/ Utilisez cette commande Artisan pour lancer le processus de seeding :

php artisan db:seed

Étape 8 : Test et Débogage

1. **Testez** toutes les fonctionnalités en vous connectant avec différents rôles. Vérifiez que les restrictions d'accès fonctionnent comme prévu.
2. **Déboguez** tout problème rencontré, en vous assurant que les middlewares sont correctement appliqués et que la logique de vos contrôleurs est correcte.

Après avoir configuré les middlewares et les routes, testez votre application en accédant aux différentes routes avec des utilisateurs ayant différents rôles pour vous assurer que les restrictions d'accès fonctionnent comme prévu. Corrigez tout problème de logique ou de configuration des middlewares rencontré lors des tests.

Cet exemple détaillé montre comment utiliser les middlewares dans Laravel pour contrôler l'accès aux différentes parties de votre application en fonction du rôle de l'utilisateur, offrant ainsi une sécurité renforcée et une meilleure gestion des permissions.