

I-Contrôleurs Invokables et Contrôleurs Ressources en Laravel

Dans ce cours, nous allons explorer deux types de contrôleurs en Laravel :

1. Le **contrôleur invokable** (ou contrôleur à méthode unique).
2. Le **contrôleur resource** (ou contrôleur CRUD).

Nous verrons comment les créer, les configurer, et comment définir les routes associées.

0. Contrôleur Invokable

A. Qu'est-ce qu'un Contrôleur Invokable ?

Un **contrôleur invokable** est un contrôleur qui ne contient qu'une seule méthode publique nommée `__invoke()`.

- On l'utilise lorsque vous n'avez besoin que d'une seule action dans votre contrôleur.
- Au lieu d'avoir plusieurs méthodes (`index`, `show`, `create`, etc.), vous n'en avez qu'une seule qui sera appelée automatiquement lorsque vous invoquerez ce contrôleur dans la route.

B. Comment le Créer ?

Pour créer un contrôleur invokable, utilisez la commande artisan suivante :

```
php artisan make:controller HelloWorldController --invokable
```

Cela va générer un fichier de contrôleur à l'emplacement `app/Http/Controllers/HelloWorldController.php` avec une seule méthode `__invoke()`.

C. Exemple de Code

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HelloWorldController extends Controller
{
    /**
     * Handle the incoming request.
     */
    public function __invoke()
    {
        return 'Hello World from an invokable controller!';
    }
}
```

```
}  
}
```

D. Définir la Route

Pour utiliser ce contrôleur invocable dans une route, vous pouvez simplement faire :

```
use App\Http\Controllers\HelloWorldController;  
use Illuminate\Support\Facades\Route;  
  
Route::get('/hello', HelloWorldController::class);
```

- Ici, Laravel sait qu'il doit faire appel à la méthode `__invoke()` de votre `HelloWorldController` quand un utilisateur visite l'URL `/hello`.

1. Créer un Contrôleur Resource

A. Qu'est-ce qu'un Contrôleur Resource ?

Un **contrôleur resource** est un type de contrôleur qui regroupe automatiquement les méthodes nécessaires pour les opérations CRUD (Create, Read, Update, Delete) sur une ressource.

- Il comprend généralement les méthodes `index`, `create`, `store`, `show`, `edit`, `update`, `destroy`.
- Lorsque vous utilisez le routage resource, Laravel génère pour vous les routes associées à ces méthodes.

B. Comment le Créer ?

Pour créer un contrôleur resource, exécutez la commande artisan :

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

Ceci génère un contrôleur appelé `ArticleController` avec toutes les méthodes standard pour la gestion d'articles.

2. Définir les Routes pour un Contrôleur Resource

A. Routes CRUD Manuelles

Si vous utilisez le raccourci `Route::resource()`, cela est équivalent à :

```
Route::get('/articles', [ArticleController::class, 'index'])->name('articles.index');
Route::get('/articles/create', [ArticleController::class, 'create'])->name('articles.create');
Route::post('/articles', [ArticleController::class, 'store'])->name('articles.store');
Route::get('/articles/{id}', [ArticleController::class, 'show'])->name('articles.show');
Route::get('/articles/{id}/edit', [ArticleController::class, 'edit'])->name('articles.edit');
Route::put('/articles/{id}', [ArticleController::class, 'update'])->name('articles.update');
Route::delete('/articles/{id}', [ArticleController::class, 'destroy'])->name('articles.destroy');
```

Chaque route est associée à la méthode correspondante du contrôleur.

B. Raccourci : `Route::resource()`

Pour simplifier, Laravel propose la méthode `Route::resource()`. Elle crée automatiquement toutes les routes CRUD avec un seul appel :

```
Route::resource('articles', ArticleController::class);
```

Derrière, cela génère exactement les mêmes routes que si vous les aviez listées une par une. La différence, c'est qu'il n'y a plus qu'une seule ligne de code à maintenir.

Note : Pour personnaliser les noms de routes générées par `Route::resource()`, vous pouvez passer un tableau de paramètres, mais la configuration par défaut convient à la plupart des cas.

3. Afficher la Liste des Articles

A. La Route

Imaginons que nous souhaitons afficher la liste de tous les articles via un contrôleur de ressource nommé `ArticleController`. Nous pouvons définir une route GET :

```
Route::get('/articles',
[\App\Http\Controllers\ArticleController::class, 'index'])->name('all');
```

Ici, nous avons :

- L'URL `/articles` qui pointe vers la méthode `index()`.
- Un nom de route `all` pour simplifier les appels dans les vues.

B. Méthode `index()` dans le Contrôleur

Dans le contrôleur `ArticleController`, la méthode `index()` peut ressembler à ceci :

```
namespace App\Http\Controllers;

use App\Models\Article;
use Illuminate\Http\Request;
```

```

class ArticleController extends Controller
{
    // ...

    public function index()
    {
        $articles = Article::paginate(10); // Par exemple, 10
//articles par page
        return view('articles.index', compact('articles'));
    }

    // ...
}

```

C. Appeler la Route Depuis la Vue

Pour faire un lien dans votre vue, utilisez le nom de la route défini avec

```

->name('all') :

<a href="{{ route('all') }}">Liste des articles</a>

```

Lorsque l'utilisateur clique sur ce lien, il est redirigé vers l'URL `/articles`.

4. Exemples Concrets : Routes et Contrôleurs dans `web.php`

Voici un fichier `web.php` plus complet pour illustrer l'utilisation de différents contrôleurs :

```

<?php

use App\Http\Controllers\HelloWorldController;
use App\Http\Controllers\ProduitController;
use App\Http\Controllers\RproductController;
use App\Http\Controllers\ArticleController;
use Illuminate\Support\Facades\Route;

Route::get('/', [ProduitController::class, 'home']);

// Exemple : Récupérer des produits par catégorie
Route::get('/produits/{cat}',
[ProduitController::class, 'getProdByCat']);

// Contrôleur invokable
Route::get('/hello', HelloWorldController::class);

// Contrôleur resource pour gérer les produits
Route::resource('produits', RproductController::class);

// Route spécifique pour lister des articles dans RproductController
Route::get('/articles', [RproductController::class, 'index'])->name('index');

```

```
// Contrôleur resource pour gérer les articles
Route::resource('articles', ArticleController::class);
```

Attention : Si vous définissez manuellement `Route::get('/articles', [RproductController::class, 'index'])` et en même temps `Route::resource('articles', ArticleController::class)`, vérifiez qu'il n'y a pas de conflit d'URL ou de duplication de route. Généralement, on sépare clairement les ressources pour éviter la confusion.

5. Exemple de Contrôleur Resource : `RproductController`

```
namespace App\Http\Controllers;

use App\Models\Produit;
use Illuminate\Http\Request;

class RproductController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        // On récupère les produits paginés par 3
        $produits = Produit::paginate(3);

        // On renvoie la vue home.blade.php
        // en lui passant la liste paginée des produits
        return view('home', ['products' => $produits]);
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        // Retourner une vue pour créer un produit
        // return view('produits.create');
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        // Logique pour enregistrer un nouveau produit en base
    }

    /**
     * Display the specified resource.
     */
}
```

```

public function show(string $id)
{
    // Afficher un produit précis
    // $produit = Produit::findOrFail($id);
    // return view('produits.show', compact('produit'));
}

/**
 * Show the form for editing the specified resource.
 */
public function edit(string $id)
{
    // Retourner une vue avec le produit à modifier
}

/**
 * Update the specified resource in storage.
 */
public function update(Request $request, string $id)
{
    // Mettre à jour un produit dans la base
}

/**
 * Remove the specified resource from storage.
 */
public function destroy(string $id)
{
    // Supprimer un produit de la base
}
}

```

Dans cet exemple :

- La méthode `index()` est utilisée pour afficher la liste des produits (avec pagination).
- Les autres méthodes (`create`, `store`, `show`, etc.) sont prêtes à être implémentées dès que vous souhaitez gérer la création, l’affichage, la mise à jour et la suppression de produits.

Conclusion

- **Le contrôleur invokable** vous permet de simplifier lorsque vous n’avez qu’une seule méthode à définir.
- **Le contrôleur resource** est idéal pour gérer du CRUD complet grâce à ses méthodes prédéfinies et son routage simplifié via `Route::resource()`.

En comprenant ces deux approches, vous pouvez facilement structurer votre application Laravel pour répondre à des besoins simples (une seule action) ou plus complexes (toutes les actions CRUD). L’important est d’adopter la pratique qui correspond le mieux à votre flux de développement et à la clarté de votre code.

II-Formulaire de Création d'Articles avec Validation, Téléchargement d'Images et Messages Flash

Dans ce module pratique, nous allons créer un formulaire permettant d'ajouter un article avec son titre, son contenu, et une image associée. Nous verrons également comment gérer la validation via une **Form Request personnalisée**, le **téléchargement d'images**, et l'affichage de **messages flash** pour informer l'utilisateur du succès ou de l'échec de l'opération.

1. Méthode `create()` dans le Contrôleur

Nous partons du principe que vous avez déjà un contrôleur (par exemple `ArticleController`). Pour afficher un formulaire de création d'articles, il est courant de créer une méthode `create()` :

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

- La route correspondante peut être par exemple :

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

- Cela renverra l'utilisateur vers une vue nommée `addarticle.blade.php`, dans laquelle nous allons construire un formulaire.
-

2. Création du Formulaire dans la Vue

Dans votre fichier `resources/views/addarticle.blade.php`, vous pouvez créer un formulaire HTML. Voici un exemple **basique** :

```
<!-- resources/views/addarticle.blade.php -->

@extends('layouts.app') <!-- Si vous utilisez un layout global -->

@section('content')
    <h1>Ajouter un Article</h1>

    {!!-- Inclure le message flash de succès ou d'erreur, si vous
    l'avez déjà configuré --!!}
    @include('incs.flash')

    {!!-- Afficher les erreurs de validation pour chaque champ --!!}
```

```

@if ($errors->any())
    <div style="color:red;">
        <ul>
            @foreach ($errors->all() as $erreur)
                <li>{{ $erreur }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Formulaire pour ajouter un article -->
<form action="{{ route('articles.store') }}" method="POST"
enctype="multipart/form-data">
    @csrf

    <div>
        <label for="title">Titre :</label>
        <input type="text" name="title" id="title" value="{{
old('title') }}">
        @error('title')
            <div style="color:red;">{{ $message }}</div>
        @enderror
    </div>

    <div>
        <label for="content">Contenu :</label>
        <textarea name="content" id="content">{{ old('content')
}}</textarea>
        @error('content')
            <div style="color:red;">{{ $message }}</div>
        @enderror
    </div>

    <div>
        <label for="image">Image :</label>
        <input type="file" name="image" id="image">
        @error('image')
            <div style="color:red;">{{ $message }}</div>
        @enderror
    </div>

    <button type="submit">Enregistrer</button>
</form>
@endsection

```

Points importants :

- @csrf est obligatoire pour générer un token CSRF et ainsi protéger votre application contre les attaques de type CSRF
- enctype="multipart/form-data" est nécessaire pour permettre le **téléchargement de fichiers** (ici, l'image).

Comprendre la Protection CSRF (@csrf)

Qu'est-ce qu'une attaque CSRF ?

CSRF (Cross-Site Request Forgery) est une attaque où un utilisateur authentifié sur un site peut être incité à envoyer une requête (généralement POST) à ce site **sans le vouloir**, car l'attaquant exploite le fait que les cookies de session sont automatiquement inclus dans les requêtes sortantes du navigateur.

Exemple concret :

1. Alice est connectée à son compte bancaire sur `www.exemplebanque.com` (solde : 1000 euros).
2. Eve, l'attaquant, crée une page web malveillante `www.malware.com` qui contient un formulaire caché pour transférer 500 euros vers son propre compte, en envoyant un POST vers `www.exemplebanque.com/transfer`.
3. Alice clique sur un lien qui la redirige (inconsciemment) vers `www.malware.com`. Le formulaire malveillant est soumis automatiquement en arrière-plan, et le navigateur envoie les cookies de session d'Alice à `www.exemplebanque.com`.
4. Résultat : 500 euros sont transférés du compte d'Alice au compte d'Eve **sans consentement** d'Alice.

Comment se protéger de la CSRF dans Laravel ?

Laravel ajoute un **jeton CSRF** dans chaque formulaire via la directive `@csrf`. Le framework vérifie ce jeton côté serveur pour s'assurer que la requête vient bien de l'application elle-même et non d'un site malveillant.

Ainsi, dans **tous** vos formulaires Blade, n'oubliez jamais d'inclure :

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

Ce token est vérifié automatiquement par un middleware Laravel et empêche toute requête non légitime dépourvue de ce token.

3. Validation via une Form Request Personnalisée

Au lieu de valider directement dans le contrôleur, Laravel propose les **Form Requests** pour avoir un code plus clair et réutilisable. Créons-en une pour la création d'un article :

```
php artisan make:request AddArticleRequest
```

Cela va générer un fichier `App\Http\Requests\AddArticleRequest.php`. Vous pouvez y définir les **règles** de validation et les **messages** d'erreur :

```
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class AddArticleRequest extends FormRequest
{
    public function authorize()
    {
        // Mettre à true si tout le monde est autorisé à soumettre ce
        //formulaire
        // ou définir une logique d'autorisation selon vos besoins.
        return true;
    }

    public function rules()
    {
        return [
            'title' => 'required|min:5',
            'content' => 'required|min:100',
            'image' => 'required|image|max:2048',
            // Par exemple, obliger l'utilisateur à charger une image
            //de max 2 Mo
        ];
    }

    public function messages()
    {
        return [
            'title.required' => 'Le titre est obligatoire.',
            'title.min' => 'Le titre doit comporter au moins 5
caractères.',
            'content.required' => 'Le contenu est obligatoire.',
            'content.min' => 'Le contenu doit contenir au moins
100 caractères.',
            'image.required' => 'Veuillez sélectionner une image.',
            'image.image' => 'Le fichier doit être une image
(jpeg, png, etc.).',
            'image.max' => 'L'image ne doit pas dépasser 2 Mo.',
        ];
    }
}
```

Note : Vous pouvez ajouter d'autres règles, comme un format d'e-mail pour un champ `email`, ou une expression régulière pour un champ `name`, etc.

4. Méthode `store()` dans le Contrôleur

Pour gérer la soumission du formulaire, nous allons injecter notre **Form Request** dans la méthode `store()`. Exemple :

```
use App\Http\Requests\AddArticleRequest;
use App\Models\Article;

public function store(AddArticleRequest $request)
{
    // 1. Validation des champs
    // (elle est déjà effectuée automatiquement, mais on peut
    forcer le check)
    $request->validated();

    // 2. Récupérer les valeurs des champs
    $title = $request->input('title');
    $content = $request->input('content');

    // 3. Gérer l'upload de l'image
    // getClientOriginalName() => récupère le nom original du
    //fichier
    $imageName = $request->file('image')->getOriginalName();

    // 4. Créer et sauvegarder un nouvel article en base de données
    $article = new Article();
    $article->titre = $title;
    $article->contenu = $content;
    $article->image = $imageName;
    $article->save();

    // 5. Déplacer l'image dans le dossier public/images
    $request->file('image')->move(public_path('images'), $imageName);

    // 6. Redirection avec un message flash de succès
    return back()->with('success', 'Vous avez ajouté un article avec
    succès !');
}
```

Pourquoi un Message Flash ?

- Les **messages flash** sont très utiles pour informer l'utilisateur d'une réussite ou d'une erreur après une action (ajout, mise à jour, suppression, etc.).
- Ces messages ne persistent qu'une seule requête HTTP, puis disparaissent (idéal pour afficher un message de confirmation).

5. Affichage des Erreurs dans la Vue

Dans la vue `addarticle.blade.php`, nous avons déjà ajouté :

```
@error('title')
    <div style="color:red;">{{ $message }}</div>
@enderror
```

- Ceci permet d'afficher un message d'erreur précis pour le champ `title`.
 - De même pour `content` et `image`.
-

6. Gestion de l'Upload d'Images

A. Dossier de Destination

Dans l'exemple ci-dessus, nous avons stocké l'image dans `public_path('images')`, c'est-à-dire `public/images`.

- Assurez-vous que ce dossier existe. S'il n'existe pas, créez-le manuellement.
- Les fichiers y seront accessibles via l'URL `http://votre-site/images/nomDeFichier.jpg`.

B. Accéder à l'Image dans la Vue

Pour afficher l'image dans une vue, vous pouvez faire :

```

```

7. Création d'un Dossier `incs` pour les Messages Flash

Pour éviter de répéter le code d'affichage du message flash dans toutes les vues, on peut créer un **fichier partiel** :

resources/views/incs/flash.blade.php :

```
@if($message = Session::get('success'))
    <div class="alert alert-success alert-block">
        <button type="button" class="close" data-dismiss="alert">X</button>
        <strong>{{ $message }}</strong>
    </div>
@endif

{{-- On peut gérer d'autres types de messages, comme errors, warning, info, etc. --}}
```

8. Inclure ce Message Flash + Scripts Bootstrap dans la Vue

Ensuite, dans vos vues principales (par exemple, `addarticle.blade.php` ou toute autre vue), vous pouvez inclure :

```
<div class="container justify-content-center mt-3">
  @include('incs.flash')
</div>
```

Si vous voulez les styles Bootstrap, n'oubliez pas d'inclure :

```
<!-- Fichiers CSS de Bootstrap (à mettre souvent dans le layout
global) -->
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.5.0/dist/css/bootstrap
.min.css">

<!-- Inclure jQuery -->
<script src="https://code.jquery.com/jquery-
3.5.1.slim.min.js"></script>

<!-- Inclure Popper.js et le JS de Bootstrap -->
<script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.mi
n.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@4.5.0/dist/js/bootstrap.m
in.js"></script>
```

Vous pouvez alors bénéficier de l'apparence Bootstrap pour vos messages flash, vos formulaires, etc.

Récapitulatif Final

1. **Route & Méthode `create()`** : Permet d'afficher la vue de création d'un article.
2. **Formulaire HTML** : On y ajoute `@csrf`, on gère le champ de fichier (`enctype="multipart/form-data"`) et on affiche les erreurs de validation.
3. **Form Request Personnalisée** (`AddArticleRequest`) : Gère la logique de validation pour plus de clarté.
4. **Méthode `store()`** : Récupère les données, valide, enregistre en base, déplace l'image dans `public/images`, et redirige l'utilisateur avec un message flash.
5. **Affichage des Erreurs** : `@error('champ')` ... `@enderror` pour chaque champ concerné.
6. **Upload d'Image** : Utiliser la méthode `move()` pour enregistrer le fichier et stocker son nom en base.
7. **Messages Flash** : Centraliser l'affichage via `incs.flash`.
8. **Styles et Scripts** : Intégrer Bootstrap (ou tout autre framework) pour le style et les interactions.

Avec ce flux de travail, vous avez désormais un mini-système de publication d'articles capable de gérer la validation et le téléchargement d'images. Libre à vous de personnaliser et d'étendre les fonctionnalités (pagination, édition, suppression, etc.) pour en faire un véritable module de gestion de contenu.

III-Atelier 8 :Énoncé Pratique : Ajout d'un Nouveau Produit sur un Site E-commerce

L'objectif de cet exercice est de mettre en place une fonctionnalité **AddProduct** permettant à un utilisateur d'ajouter un nouveau produit à la base de données. Vous aurez besoin de :

1. **Une route** pour afficher le formulaire d'ajout (méthode `create()`).
2. **Une route** pour enregistrer en base le produit (méthode `store()`).
3. **Un contrôleur** (ici `RproductController`) pour gérer la logique d'enregistrement.
4. **Une classe de Form Request** (`AddProductRequest`) pour valider les champs du formulaire.
5. **Un message flash** pour informer l'utilisateur en cas de succès.
6. **Une vue** (`addproduit.blade.php`) contenant un formulaire avec en-tête `multipart/form-data` pour l'upload d'images.

Ci-dessous, vous trouverez le code déjà mis en place ainsi que les explications pour chacun des éléments. Veuillez à bien relier toutes ces parties pour que la fonctionnalité **AddProduct** soit complète et fonctionnelle.

1. Le Fichier de Routes : `web.php`

```
<?php

use App\Http\Controllers\HelloWorldController;
use App\Http\Controllers\ProduitController;
use App\Http\Controllers\RproductController;
use Illuminate\Support\Facades\Route;

/*
|-----
|-----
| Web Routes
|-----
|-----
|
| Ici, on déclare les routes de notre application.
| Elles sont automatiquement chargées par le RouteServiceProvider.
|
*/

Route::get('/', [ProduitController::class, 'home']);

// Route pour récupérer des produits selon une catégorie (exemple)
Route::get('/produits/{cat}', [ProduitController::class,
'getProdByCat']);

// Contrôleur invokable
Route::get('/hello', HelloWorldController::class);

/**
```

```

    * Les routes CRUD (Create, Read, Update, Delete) du contrôleur
    ressource RproductController.
    * Cela génère automatiquement :
    *   - GET /produits (index)
    *   - GET /produits/create (create)
    *   - POST /produits (store)
    *   - GET /produits/{id} (show)
    *   - GET /produits/{id}/edit (edit)
    *   - PUT/PATCH /produits/{id} (update)
    *   - DELETE /produits/{id} (destroy)
    */
Route::resource('produits', RproductController::class);

/**
 * Pour plus de clarté, vous pouvez surcharger la route GET
 /produits/create
 * si vous souhaitez un nom particulier, par exemple:
 */
Route::get('/produits/create', [RproductController::class,
'create'])->name('create');

```

Explications

- `Route::resource('produits', RproductController::class);` fournit toutes les routes nécessaires au CRUD.
- La route `/produits/create` pointera vers la méthode `create()` du `RproductController`, qui renvoie la vue du formulaire.
- La soumission du formulaire (méthode POST) pointera vers `/produits`, qui déclenchera la méthode `store()` du `RproductController`.

2. Le Contrôleur : `RproductController`

```

<?php

namespace App\Http\Controllers;

use App\Http\Requests\AddProductRequest;
use App\Models\Produit;
use Illuminate\Http\Request;

class RproductController extends Controller
{
    /**
     * Affiche la liste paginée des produits (méthode index).
     */
    public function index()
    {
        // On récupère 3 produits par page
        $produits = Produit::paginate(3);

        // On renvoie la vue 'home.blade.php' avec les produits
        return view('home', ['products' => $produits]);
    }

    /**

```

```

    * Affiche le formulaire pour créer un nouveau produit.
    */
public function create()
{
    // Retourne la vue 'Addproduit.blade.php'
    return view('Addproduit');
}

/**
 * Enregistre un nouveau produit dans la base (Store).
 */
public function store(AddProductRequest $request)
{
    // 1. Valider les champs via la Form Request
    $request->validated();

    // 2. Récupérer les valeurs des champs
    $nom          = $request->input('nom');
    $prix         = $request->input('prix');
    $categorie    = $request->input('categorie');
    // getClientOriginalName() retourne le nom d'origine de
    l'image uploadée
    $image        = $request->file('image')->
    >getClientOriginalName();

    // 3. Créer un nouvel objet Produit
    $Produit = new Produit();
    $Produit->nom          = $nom;
    $Produit->prix         = $prix;
    $Produit->categorie    = $categorie;
    $Produit->image        = $image;

    // 4. Enregistrer dans la table 'produits'
    $Produit->save();

    // 5. Déplacer l'image dans le dossier public/imgs
    $request->file('image')->move(public_path('imgs'), $image);

    // 6. Retourner à la même page avec un message flash de
    //succès
    return back()->with('success', 'You have successfully added a
    new Product.');
```

```

    }

    /**
     * Affiche un produit précis (Show).
     */
    public function show(string $id)
    {
        //
    }

    /**
     * Formulaire d'édition (Edit).
     */
    public function edit(string $id)
    {
        //
    }

    /**

```

```

        * Mettre à jour un produit en base (Update).
        */
public function update(Request $request, string $id)
{
    //
}

/**
 * Supprimer un produit (Destroy).
 */
public function destroy(string $id)
{
    //
}
}

```

Explications

- Le **Form Request** `AddProductRequest` assure la validation.
- Le fichier est ensuite **déplacé** dans `public/imgs`, de sorte que les images soient directement accessibles via l'URL `/imgs/nom-de-l-image`.
- Le **message flash** est renvoyé pour informer l'utilisateur que le produit a bien été ajouté.

3. La Classe de Form Request : `AddProductRequest`

```

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class AddProductRequest extends FormRequest
{
    /**
     * Détermine si l'utilisateur est autorisé à faire cette requête.
     */
    public function authorize(): bool
    {
        // Ici, on autorise tout le monde.
        return true;
    }

    /**
     * Règles de validation
     */
    public function rules(): array
    {
        return [
            'nom' => 'required|min:5',
            'prix' => 'required',
            'categorie' => 'required|min:5',
        ];
    }
}

```

```

/**
 * Messages d'erreur personnalisés
 */
public function messages()
{
    return [
        'nom.required'      => 'name is required',
        'nom.min'           => 'too short enter more',
        'prix.required'     => 'price is required',
        'categorie.required' => 'category is required',
        'categorie.min'     => 'too short enter more',
    ];
}
}

```

Explications

- Les **règles** indiquent que :
 - Le champ `nom` est obligatoire et doit avoir au moins 5 caractères.
 - Le champ `prix` est obligatoire.
 - Le champ `categorie` est obligatoire et doit avoir au moins 5 caractères.
- Les **messages** donnent des retours personnalisés pour l'utilisateur.

4. Le Message Flash : `flash.blade.php`

```

@if($message = Session::get('success'))
    <div class="alert alert-success alert-block">
        <button type="button" class="close" data-
dismiss="alert">X</button>
        <strong>{{ $message }}</strong>
    </div>
@endif

```

Explications

- Ce fichier partiel peut être inclus dans vos vues pour afficher un message de type **"success"** lorsque l'utilisateur a réalisé une action sans erreur.
- Le message stocké dans la session via `->with('success', '...')` est récupéré ici grâce à `Session::get('success')`.

5. La Vue : `addproduit.blade.php`

```

@extends('Master_page')
@section('title', 'add Produits')

@section('content')
    <h2>Ajouter un nouveau produit :</h2>

    <div class="container justify-content-center mt-3">
        @include('incs.flash') <!-- Inclusion du message flash -->
    </div>

```

```

</div>

<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <div class="card">
        <div class="card-header">{{ __('Ajouter un nouveau
produit') }}</div>

        <div class="card-body">
          <form method="POST" action="/produits"
enctype="multipart/form-data">
            @csrf

            <div class="form-group">
              <label for="nom">{{ __('Nom du produit')
}}</label>
              <input id="nom" type="text" class="form-
control" name="nom">
              @error('nom')
                <div style="color:red;">{{ $message
}}</div>
              @enderror
            </div>

            <div class="form-group">
              <label for="prix">{{ __('Prix du
produit') }}</label>
              <input id="prix" type="text" class="form-
control" name="prix">
              @error('prix')
                <div style="color:red;">{{ $message
}}</div>
              @enderror
            </div>

            <div class="form-group">
              <label for="categorie">{{ __('Catégorie
du produit') }}</label>
              <input id="categorie" type="text"
class="form-control" name="categorie">
              @error('categorie')
                <div style="color:red;">{{ $message
}}</div>
              @enderror
            </div>

            <div class="form-group">
              <label for="image">{{ __('Image du
produit') }}</label>
              <input id="image" type="file"
class="form-control-file" name="image">
              {{-- Pas de règle de validation pour
l'image dans l'exemple,
mais vous pouvez en rajouter. --}}
            </div>

            <div class="form-group mb-0">
              <button type="submit" class="btn btn-
primary">
                {{ __('Ajouter le produit') }}

```

```

        </button>
    </div>
</form>
</div>
</div>
</div>
</div>
</div>
</div>
</div>
@endsection

<!-- Inclure jQuery -->
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<!-- Inclure les fichiers JavaScript de Bootstrap 4 -->
<script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@4.5.0/dist/js/bootstrap.min.js"></script>

```

Explications

- Le formulaire doit inclure `@csrf` (pour la protection CSRF) et `enctype="multipart/form-data"` si vous souhaitez **téléverser une image**.
 - On utilise `@error('nom')` pour afficher l'erreur liée au champ `nom`.
-

Conclusion et Consignes

1. **Vérifiez** que votre route `/produits/create` est bien accessible et renvoie la vue `addproduit.blade.php`.
2. **Saisissez** un produit (nom, prix, catégorie) et **choisissez** une image à importer.
3. **Soumettez** le formulaire.
 - Si tout se passe bien, vous devriez être **redirigé** vers la même page et voir apparaître un **message flash** de succès.
 - Un **nouveau produit** doit alors exister en base de données.
 - L'**image** doit être présente dans le dossier `public/imgs`.
4. **Vérifiez** les **messages d'erreur** si vous laissez un champ vide ou si vous ne respectez pas la taille minimum requise (5 caractères pour le nom et la catégorie).

Vous aurez ainsi un module **AddProduct** fonctionnel, avec **validation**, **upload d'images**, et **messages flash**. Cela constitue une base solide pour enrichir votre site e-commerce en y ajoutant la gestion (mise à jour, suppression) et l'affichage détaillé de chaque produit.