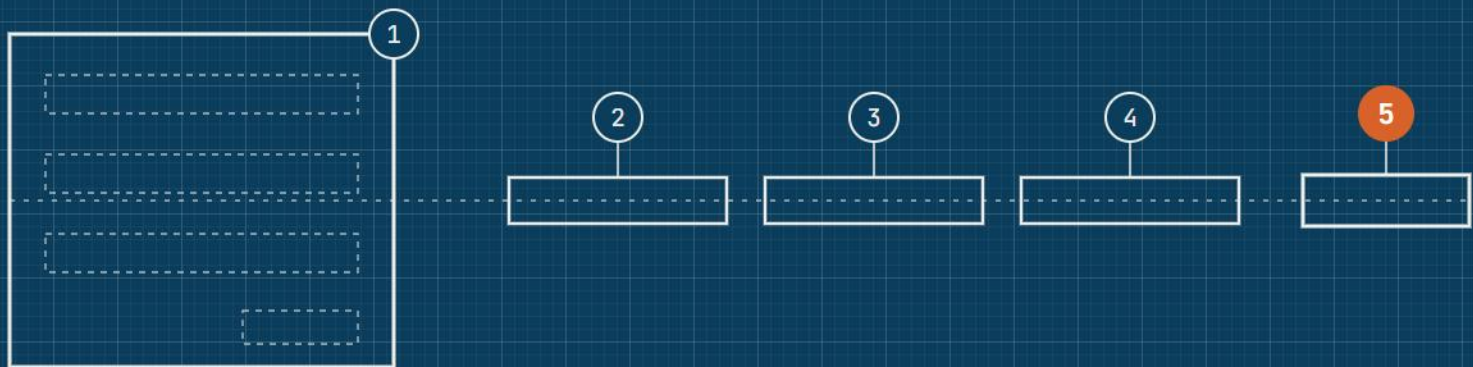


Bloc Laravel · chapitre 15

Formulaires : créer et valider

Nadia, qui tient une friterie à Wavre, ajoute elle-même un produit à sa carte depuis un formulaire. Vous écrivez ce formulaire, puis vous vérifiez chaque champ avant de l'enregistrer.



Le dépôt friagerie-26 s'installe une fois, en sept commandes

Sur github.com/opmvpc/friagerie-26, créez votre copie avec le bouton Fork. Dans un terminal ouvert dans le dossier de Herd ou de Laragon :

terminal

```
$ git clone https://github.com/VOTRE-COMPTE/friagerie-26.git
$ cd friagerie-26
$ composer install
$ cp .env.example .env
$ php artisan key:generate
$ php artisan migrate --seed
$ npm install
```

La base reçoit la carte de Nadia : quatre catégories, huit allergènes et seize produits. Lancez ensuite `composer run dev`, laissez son terminal ouvert, et ouvrez <http://friagerie-26.test>.

routes/web.php déclare les pages déjà écrites de la friagerie

Les pages sont écrites avec ce que vous connaissez des chapitres 7 à 12 : la carte, la fiche d'un produit, une catégorie et le guide de style.

routes/web.php

```
① 1 Route::redirect('/', '/carte');  
2  
3 Route::get('/carte', [ProductController::class, 'index'])->name('products.index');  
4 Route::get('/produits/{id}', [ProductController::class, 'show'])② ->whereNumber('id')->name('products.show');  
5  
6 Route::get('/categories/{id}', [CategoryController::class,  
    'show'])->whereNumber('id')->name('categories.show');  
7  
③ 8 Route::view('/composants', 'styleguide')->name('styleguide');
```

- ① L'accueil renvoie vers la carte.
- ② Refuse un id qui n'est pas un nombre.
- ③ Le guide de style, sans contrôleur.

Quatre composants servent aux formulaires : `<x-label>`, `<x-input>`, `<x-select>` et `<x-textarea>`. `/composants` les montre normaux puis en erreur.

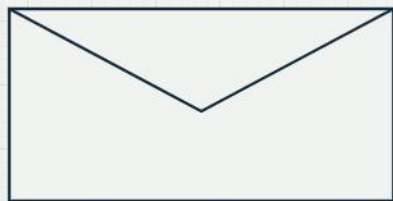
Une route GET retire un produit : qui d'autre que vous peut la suivre?

En binôme, cinq minutes. Un collègue de Nadia veut retirer un produit d'un simple clic. Il ajoute cette route, et un lien « Retirer » sur chaque fiche :

routes/web.php

```
1 | Route::get('/produits/{id}/retirer', fn (int $id) =>  
    Product::findOrFail($id)->delete());
```

Le lien fonctionne : `/produits/16/retirer` affiche 1, et le café a disparu de la carte. Qu'arrive-t-il quand un moteur de recherche, ou le préchargement du navigateur, suit ce lien ?



GET /produits/16/retirer



Le formulaire d'ajout demande deux routes : un GET qui l'affiche, un POST qui le reçoit

Nadia veut ajouter la viande à sa carte. Ajoutez les deux routes sous la route /carte, avant /produits/{id}, comme le demande le piège de l'ordre du chapitre 7.

routes/web.php

```
1 | Route::get('/produits/nouveau', [ProductController::class,  
  | 'create'])->name('products.create');  
2 | Route::post('/produits', [ProductController::class, 'store'])->name('products.store');
```

app/Http/Controllers/ProductController.php

```
1 | public function create(): View  
2 | {  
3 |     $categories = Category::orderBy('position')->get();  
4 |  
5 |     return view('products.create', ['categories' => $categories]);  
6 | }
```

La méthode create() passe les catégories à la vue, dans l'ordre de la carte.

Le formulaire envoie ses champs en POST à la route products.store

Créez la vue avec `php artisan make:view products.create`, puis remplacez tout son contenu par le formulaire :

resources/views/products/create.blade.php

```
1 | <x-layouts.app title="Ajouter un produit">
2 |     <div class="mx-auto max-w-2xl">
3 |         <h1 class="mb-8 text-3xl font-semibold tracking-tight">Ajouter un produit</h1>
4 |
5 |         <form ① method="POST" ② action="{{ route('products.store') }}" class="space-y-6 rounded-xl border border-zinc-200
        bg-white p-6 shadow-xs sm:p-8 dark:border-zinc-800 dark:bg-zinc-900">
6 |             ③ @csrf
7 |
8 |             <div>
9 |                 <x-label for="name">Nom</x-label>
10 |                 <x-input name="name" />
11 |             </div>
12 |             ...
```

- ① Le verbe de la route qui reçoit les champs.
- ② L'adresse de cette route, écrite par son nom.
- ③ Le jeton du chapitre 9.

La suite du formulaire ajoute la catégorie, le prix, la description et le bouton.

Chaque <option> envoie le numéro de la catégorie, et non son nom

Suite du même fichier : la catégorie se choisit dans une liste, remplie par les catégories que create() a passées.

resources/views/products/create.blade.php

```
1      <div>
2          <x-label for="category_id">Catégorie</x-label>
3          <x-select name="category_id">
4              <option value="">Choisissez une catégorie</option>
5              @foreach ($categories as $category)
6                  <option value="{{ $category->id }}">{{ $category->name }}</option>
7              @endforeach
8          </x-select>
9      </div>
10     ...
11     <x-button type="submit">Ajouter le produit</x-button>
12     ...
```

La colonne category_id attend ce numéro. Sans type="submit", le composant button écrit type="button", et le clic n'envoie rien.

store() crée le produit avec les seuls champs que only() choisit

Au chapitre 9, input() lisait un champ de la requête. La méthode only() renvoie un tableau des seuls champs cités.

app/Http/Controllers/ProductController.php

```
1 public function store(Request $request): RedirectResponse
2 {
3     $product = Product::create(① $request->only(['name', 'category_id', 'price',
4         'description']));
5
6 ② return redirect()->route('products.show', $product->id);
7 }
```

- ① Un tableau des quatre champs du formulaire.
- ② La redirection du chapitre 7 ouvre la fiche.

create() remplit les colonnes de #[Fillable], comme au chapitre 10. Envoyé avec Viandelle, Snacks et 2.80, le formulaire arrive sur /produits/17, la fiche du nouveau produit.

Envoyé vide, le formulaire provoque une erreur 500 de la base

Envoyez maintenant le formulaire sans rien remplir.

navigateur · erreur 500

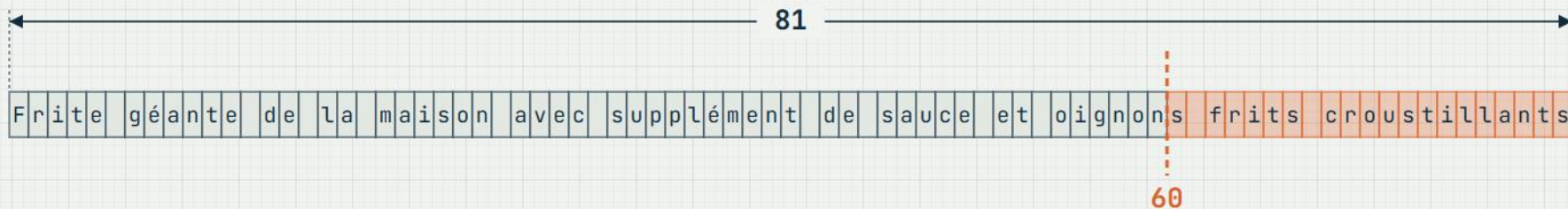
```
Illuminate\Database\QueryException  
SQLSTATE[23000]: Integrity constraint violation: 19 NOT NULL constraint failed:  
products.category_id (Connection: sqlite, ..., SQL: insert into "products" ("name",  
"category_id", "price", "description", "updated_at", "created_at") values (?, ?, ?, ?,  
2026-10-07 15:27:23, 2026-10-07 15:27:23))
```

Laravel transforme chaque champ vide en null. La colonne `category_id` n'accepte pas null, et la base refuse la ligne. Le message ne cite que la première colonne fautive, mais `name` et `price` manquent aussi.

NON CONFORME

La base accepte un nom de 81 caractères et un prix de zéro

Tapez le nom « Frite géante de la maison avec supplément de sauce et oignons frits croustillants », choisissez Frites et un prix de 0. /produits/18 affiche ces 81 caractères à 0,00 €.



La migration déclare `string('name', 60)`, mais SQLite ne vérifie pas cette longueur, et aucune colonne n'interdit un prix de zéro.

validate() reçoit, pour chaque champ, une liste de conditions

Vérifier chaque champ reçu avant de s'en servir s'appelle la **validation**.

app/Http/Controllers/ProductController.php

```
1 public function store(Request $request): RedirectResponse
2 {
3     $validated = $request->validate([
4         'name' => ['required', 'string', 'max:60'],
5         'category_id' => ['required', 'integer', 'exists:categories,id'],
6         'price' => ['required', 'numeric', 'min:0.5', 'max:50'],
7         'description' => ['nullable', 'string', 'max:500'],
8     ]);
9
10    ...
11 }
```

Chaque condition s'appelle une **règle**. La suite de store() ne change pas pour l'instant.

required, numeric, exists : chaque règle vérifie une seule condition

Règle	Ce qu'elle vérifie
required	le champ est rempli
string, max:60	un texte de 60 caractères au plus
numeric, min:0.5, max:50	un nombre entre 0,5 et 50
nullable	le champ peut rester vide, et ses autres règles ne s'appliquent alors pas
integer	un nombre entier, et non un texte ou une liste de valeurs
exists:categories,id	la table categories a une ligne dont l'id vaut la valeur reçue

Pour exists, Laravel envoie `select count(*) as "aggregate" from "categories" where "id" = ?`, avec la valeur reçue à la place du ?.

Vérifiez chaque champ sur le serveur, même ceux que le navigateur contrôle déjà

Le `<select>` ne propose que quatre catégories, mais les outils de développement changent une option en deux clics. `min="0"` fait bloquer un prix négatif par le navigateur, mais une requête modifiée passe sans ce contrôle.

app/Http/Controllers/ProductController.php · extrait

```
1 |         'category_id' => ['required', ❶ 'integer', ❷ 'exists:categories,id'],  
2 |         'price' => ['required', 'numeric', ❸ 'min:0.5', 'max:50'],
```

- ❶ Refuse une liste de numéros, que exists accepterait.
- ❷ Refuse un numéro sans ligne dans categories.
- ❸ Refuse un prix négatif, même sans `min="0"`.

Quand une règle échoue, Laravel répond par une redirection 302 vers le formulaire

Ouvrez l'onglet Réseau avec F12, et envoyez de nouveau le formulaire vide. Il revient, vide, avec trois champs bordés de rouge et sans un mot d'explication.

navigateur · onglet Réseau, réponse à produits

HTTP/1.1 302 Found

...

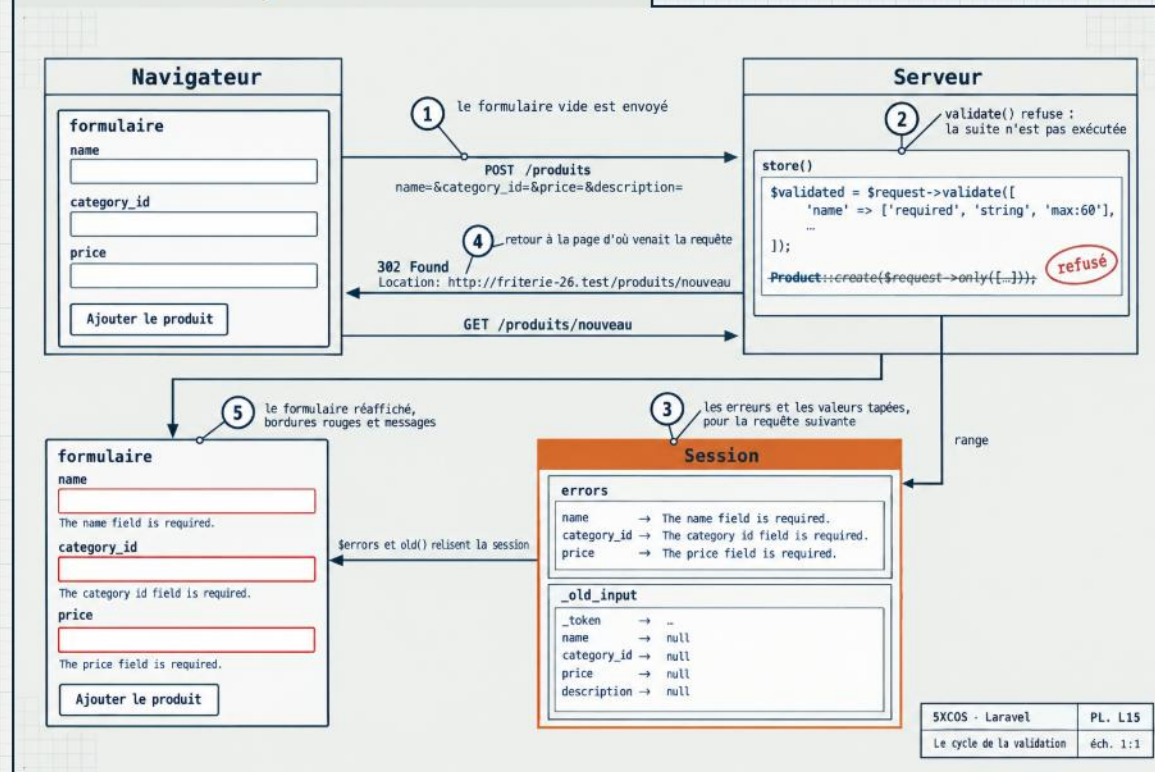
Location: http://friterie-26.test/produits/nouveau

...

Deux requêtes : produits en POST, code 302, puis nouveau en GET, code 200. `validate()` a refusé trois champs, et la suite de `store()` n'a pas été exécutée.

Laravel range les erreurs et les valeurs tapées dans la session, puis redirige

schéma • le cycle de la validation



La **session**, vue au chapitre 9, est un espace de stockage que le serveur réserve à chaque visiteur. Les erreurs et les valeurs tapées y attendent la requête suivante, celle du formulaire.

Le composant <x-input> lit \$errors et passe sa bordure au rouge

Dans chaque vue, Laravel fournit la variable `$errors`, qui contient les messages rangés dans la session. Les bordures rouges viennent des composants, qui lisent cette variable.

resources/views/components/input.blade.php · extrait

```
1  @php
2  ① $invalid = $errors->has($name);
3  @endphp
4
5  <input
6      ...
7  ② @if ($invalid) aria-invalid="true" aria-describedby="{{ $name }}-error" @endif
8      {{ $attributes->merge(['id' => $name])->class([
9          ...
10  ③ 'border-red-500 text-red-900 focus:border-red-600 focus:ring-red-600/20
    dark:border-red-400 dark:text-red-100' => $invalid,
11      ]) }}
12  >
```

- ① Vrai quand le serveur a refusé ce champ.
- ② Les attributs `aria-*` de l'exercice 14.7.
- ③ Les classes rouges, quand `$invalid` est vrai.

`@php ... @endphp` exécute quelques lignes de PHP dans le template : ici, elles rangent une condition d'affichage dans `$invalid`.

@error('name') écrit sous le champ le message du champ refusé

Le message, lui, s'écrit dans la vue, sous chaque champ. Voici le champ name :

resources/views/products/create.blade.php • extrait

```
1 |         <div>
2 |             <x-label for="name">Nom</x-label>
3 |             <x-input name="name" ① :value="old('name')" />
② 4 |             @error('name')
5 |                 <p id="name-error" class="mt-2 text-sm
text-red-600 dark:text-red-400">③ {{ $message }}</p>
6 |             @enderror
7 |         </div>
```

- ① La valeur tapée avant la redirection, ou null.
- ② Affiché seulement si name a un message.
- ③ Le message de ce champ.

Envoyez de nouveau le formulaire vide :

navigateur • /produits/nouveau

The name field is required.
The category id field is required.
The price field is required.

Quand une règle échoue, @error dit ce qui ne va pas, et old() rend ce qui avait été tapé

Pour la liste et la zone de texte, la valeur tapée revient autrement :

resources/views/products/create.blade.php · extrait

```
1         <x-select name="category_id">
2             <option value="">Choisissez une catégorie</option>
3             @foreach ($categories as $category)
4                 <option value="{{ $category->id }}" @selected(old('category_id')
5                     == $category->id)>{{ $category->name }}</option>
6             @endforeach
7         </x-select>
8         ...
9         <x-textarea name="description">{{ old('description') }}</x-textarea>
```

@selected(...) écrit selected quand la condition est vraie. Tapez Boulette et 3.50, sans catégorie : le formulaire revient avec un seul message, sous la catégorie, et garde le nom et le prix.

validate() renvoie un tableau des seules valeurs validées

Ajoutez `dd($validated);` sous l'appel à `validate()`, puis envoyez Boulette, Snacks, 3.50, sans description :

navigateur · /produits

```
array:4 [▼ // app/Http/Controllers/ProductController.php:43
    "name" => "Boulette"
    "category_id" => "2"
    "price" => "3.50"
    "description" => null
]
```

Le tableau ne contient que les champs qui ont des règles et qui figurent dans la requête, avec des valeurs qui les respectent.

#[Fillable] ne remplace pas \$validated

Retirez le `dd()`, et remplacez la ligne de `create()` :

```
app/Http/Controllers/ProductController.php
```

```
1 | $product = Product::create($validated);
```

La liste `#[Fillable]` contient `sold_out_at`, la colonne de la rupture. `$request->all()` renvoie tous les champs reçus : avec `Product::create($request->all())`, une requête qui ajoute un champ `sold_out_at` crée un produit déjà en rupture, sans qu'aucune règle l'ait examiné.

NON CONFORME

Laravel-Lang copie dans lang/ les messages du framework en français

Nadia lit « The category id field is required. ». Le paquet **Laravel-Lang** fournit les traductions de tous les messages du framework.

terminal

```
$ composer require laravel-lang/common --dev
...
Using version ^6.8 for laravel-lang/common
$ php artisan lang:add fr
...
fr.json ..... 12.22ms DONE
...
fr/validation.php ..... 30.08ms DONE
```

dossier du projet

```
lang/
├── fr/
│   ├── actions.php
│   ├── auth.php
│   ├── http-statuses.php
│   ├── pagination.php
│   ├── passwords.php
│   └── validation.php
└── fr.json
```

`fr/validation.php` contient un message par règle, `fr.json` les phrases que Laravel écrit en anglais, comme le titre de la page 404. `--dev` suffit : le site en ligne n'a besoin que des fichiers copiés dans `lang/`.

APP_LOCALE=fr passe les messages en français, sauf « category id »

Dans `.env`, passez la langue de l'application au français, et faites la même modification dans `.env.example`, son modèle versionné.

`.env`

```
1 APP_LOCALE=fr
```

`navigateur · /produits/nouveau`

Le champ nom est obligatoire.

Le champ category id est obligatoire.

Le champ prix est obligatoire.

Dans `lang/fr/validation.php`, le message de required est 'Le champ :attribute est obligatoire.'. Laravel remplace `:attribute` par le nom du champ.

Le tableau attributes donne à category_id son nom français

Le nom du champ est traduit par le tableau attributes, en bas de `lang/fr/validation.php`. Il contient déjà `name` et `price`. Pour `category_id`, absent, Laravel remplace le `_` par une espace. Ajoutez la ligne après `'body'` :

`lang/fr/validation.php`

```
1 | 'category_id' => 'catégorie',
```

Nom et catégorie vides, prix 2.80 : le formulaire revient avec deux messages, et garde le prix.

navigateur · `/produits/nouveau`

CN Friterie Chez Nadia

La carte

Ajouter un produit

Nom

Le champ nom est obligatoire.

Catégorie

Le champ catégorie est obligatoire.

Prix (€)

Description

Ajouter le produit

`with('status', ...)` place un message flash dans la session

Après un ajout, Nadia arrive sur la fiche sans confirmation. Dans `store()`, ajoutez `with()` à la redirection :

app/Http/Controllers/ProductController.php

```
1 | return redirect()->route('products.show', $product->id)->with('status', 'Le produit a  
   été ajouté.');
```

`with()` écrit la phrase dans la session, sous la clé `status`, pour la requête suivante seulement. Une valeur gardée ainsi le temps d'une requête s'appelle un **message flash**. Les erreurs et les valeurs tapées voyagent de la même façon.

@session('status') affiche le message flash sur la page qui suit la redirection

En haut de <main>, remplacez le commentaire `{{-- Le chapitre 15 affiche ici le message flash. --}}`:

resources/views/components/layouts/app.blade.php

```
① 1 @session('status')
   2     <p ② role="status" class="mb-8 rounded-lg border border-green-200 bg-green-50 px-4 py-3 text-sm font-medium
      text-green-800 dark:border-green-900 dark:bg-green-950 dark:text-green-200">
   3         {{ $value }}
   4     </p>
   5 @endsession
```

- ① Affiché seulement si la session contient la clé status.
- ② Les lecteurs d'écran annoncent le message.
- ③ La phrase rangée sous cette clé.

Ajoutez de nouveau la viande : /produits/17 s'ouvre sous un encadré vert.

navigateur · /produits/17

Le produit a été ajouté.

Rechargez la page : l'encadré a disparu, car la requête précédente a consommé le message.

La liaison de modèle fait le `findOrFail()` à votre place

Dans le dépôt, `show()` cherche le produit elle-même. Renommez le paramètre en `{product}`, retirez `whereNumber('id')`, et simplifiez `show()` :

app/Http/Controllers/ProductController.php · avant

```
1 | public function show(int $id): View
2 | {
3 |     $product = Product::findOrFail($id);
4 |
5 |     return view('products.show',
6 |         ['product' => $product]);
7 | }
```

routes/web.php

```
1 | Route::get('/produits/{product}',
   |     [ProductController::class,
   |     'show'])->name('products.show');
```

app/Http/Controllers/ProductController.php · après

```
1 | public function show(Product $product): View
2 | {
3 |     return view('products.show', ['product' =>
4 |         $product]);
5 | }
```

Le paramètre `{product}` et l'argument `$product` portent le même nom, et l'argument a pour type un modèle. Laravel cherche le produit, le passe à la méthode, ou répond 404. Ce mécanisme s'appelle la **liaison de modèle**.

Sans whereNumber, /produits/nouveau reste déclarée au-dessus de /produits/{product}

/produits/17 affiche la viandelle. /produits/999 et /produits/abc répondent 404, avec « Non trouvé », traduit par fr.json.

terminal

```
$ php artisan route:list --path=produits
POST      produits ..... products.store > ProductController@store
GET|HEAD  produits/nouveau ..... products.create > ProductController@create
GET|HEAD  produits/{product} ..... products.show > ProductController@show
```

Showing [3] routes

Déplacée sous /produits/{product}, la route du formulaire répond 404 : Laravel cherche un produit dont l'id vaut nouveau.

Deux exercices sur papier, puis le refuge du chapitre 17

Comptez 20 minutes pour ces deux exercices sur papier. Les exercices de code viennent au chapitre 17, après le chapitre 16 : vous y écrirez le site d'un refuge pour animaux, avec ce que la friterie vous a appris.

15.1 Prédire les messages (10 minutes). Pour chaque champ refusé par `store()`, écrivez le message affiché, une fois `category_id` ajouté à `attributes`.

```
1. name=Cornet+de+frites&category_id=&price=3&description=
2. name=Sauce+brasil&category_id=3&price=0.30&description=
3.
   name=MitrailletteMitrailletteMitrailletteMitrailletteMitrailletteM
   itraillette&category_id=99&price=60&description=
```

15.2 Le bon verbe (10 minutes). Pour chaque action, choisissez le verbe HTTP, et dites si elle part d'un lien ou d'un formulaire avec `@csrf`.

- 1 Afficher la carte.
- 2 Afficher le formulaire d'ajout.
- 3 Enregistrer un nouveau produit.
- 4 Corriger le prix de la Grande frite.
- 5 Marquer la Sauce lapin en rupture.
- 6 Retirer le café de la carte.