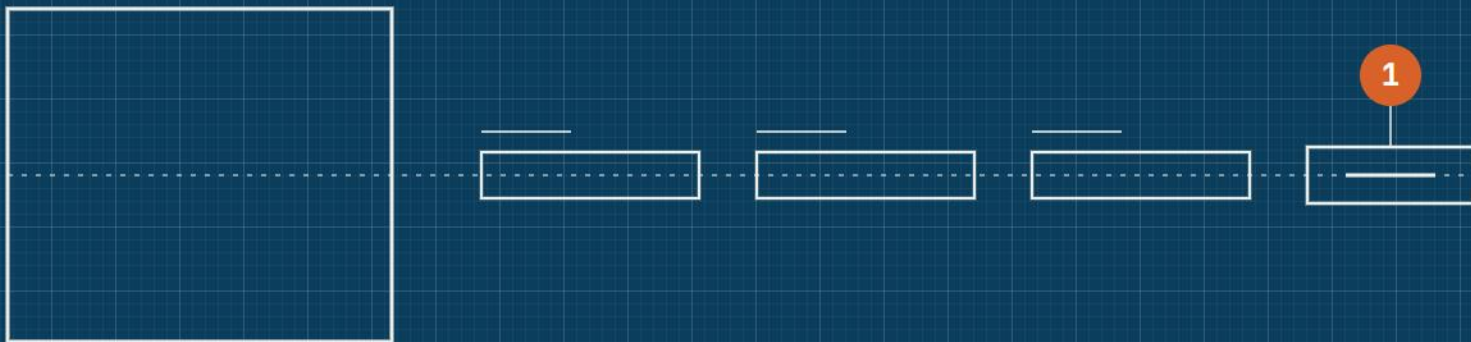


Bloc Laravel · chapitre 16

Formulaires : modifier et supprimer

Un prix augmente, un allergène a été oublié, les croquettes manquent un samedi soir : la carte de la friterie de Nadia, à Wavre, change chaque semaine.



Un formulaire écrit avec `method="PUT"` : quelle requête part ?

En binôme, cinq minutes. Un collègue de Nadia écrit le formulaire qui modifie la Grande frite. Il a retenu du chapitre 7 qu'une modification s'envoie avec le verbe PUT.

vue · le formulaire du collègue

```
1 | <form method="PUT" action="/produits/2">
2 |     @csrf
3 |     ...
4 |     <x-button type="submit">Enregistrer</x-button>
5 | </form>
```

Nadia passe le prix de 4,00 € à 4,50 € et clique sur Enregistrer. Quelle requête le navigateur envoie-t-il, et qu'affiche la page ?

Un formulaire HTML ne connaît que GET et POST

Le navigateur remplace la valeur inconnue PUT par GET, et range les champs dans l'adresse :

navigateur · barre d'adresse

/produits/2?_token=...&name=Grande+frite&category_id=1&price=4.50&...

Laravel répond par la fiche de la Grande frite, toujours à 4,00 €, sans message : rien n'est enregistré. Le jeton CSRF se retrouve aussi dans l'adresse, donc dans l'historique du navigateur.



products.edit affiche le formulaire, products.update reçoit l'envoi

Il faut deux routes, comme pour l'ajout. Ajoutez-les sous la route products.show :

routes/web.php

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

app/Http/Controllers/ProductController.php

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

Grâce à la liaison de modèle du chapitre 15, edit() reçoit le produit déjà lu dans la base, sans findOrFail(). Elle le passe à la vue avec les catégories.

edit.blade.php reprend le formulaire d'ajout avec cinq changements

terminal

```
$ php artisan make:view products.edit  
INFO View [resources/views/products/edit.blade.php] created successfully.
```

Remplacez tout le contenu du fichier, un `<div>` et une citation, par le formulaire d'ajout. Cinq changements : le titre, l'adresse d'envoi, la ligne `@method('PUT')`, le second argument de chaque `old()` et le texte du bouton.

resources/views/products/edit.blade.php

```
1 <x-layouts.app :title="'Modifier ' . $product->name">  
2     <div class="mx-auto max-w-2xl">  
3         <h1 class="mb-8 text-3xl font-semibold tracking-tight">❶ Modifier {{ $product->name }}</h1>  
4  
5         <form method="POST" action="{❷ route('products.update', $product->id) }" 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             @csrf  
❸ 7             @method('PUT')  
            ...
```

- ❶ Le titre : « Modifier » et le nom du produit.
- ❷ L'adresse d'envoi : la route `products.update`.
- ❸ Une ligne nouvelle, sous `@csrf`.

old() reçoit la valeur enregistrée en second argument

Suite du même fichier : chaque champ passe la valeur enregistrée du produit en second argument d'old(), et le bouton devient « Enregistrer ».

resources/views/products/edit.blade.php · suite

```
9      <div>
10          <x-label for="name">Nom</x-label>
11          <x-input name="name" :value="old('name', $product->name)" />
12      ...
22          <option value="{{ $category->id }}" @selected(old('category_id',
    $product->category_id) == $category->id)>{{ $category->name }}</option>
23      ...
32          <x-input name="price" type="number" step="0.10" min="0" :value="old('price',
    $product->price)" />
33      ...
40          <x-textarea name="description">{{ old('description', $product->description) }}</x-textarea>
41      ...
47          <x-button type="submit">Enregistrer</x-button>
48      ...
```

old('name', \$product->name) renvoie la valeur gardée dans la session après un refus, et sinon son second argument. Sur /produits/2/modifier, les quatre champs affichent la Grande frite.

@method('PUT') écrit un champ caché _method

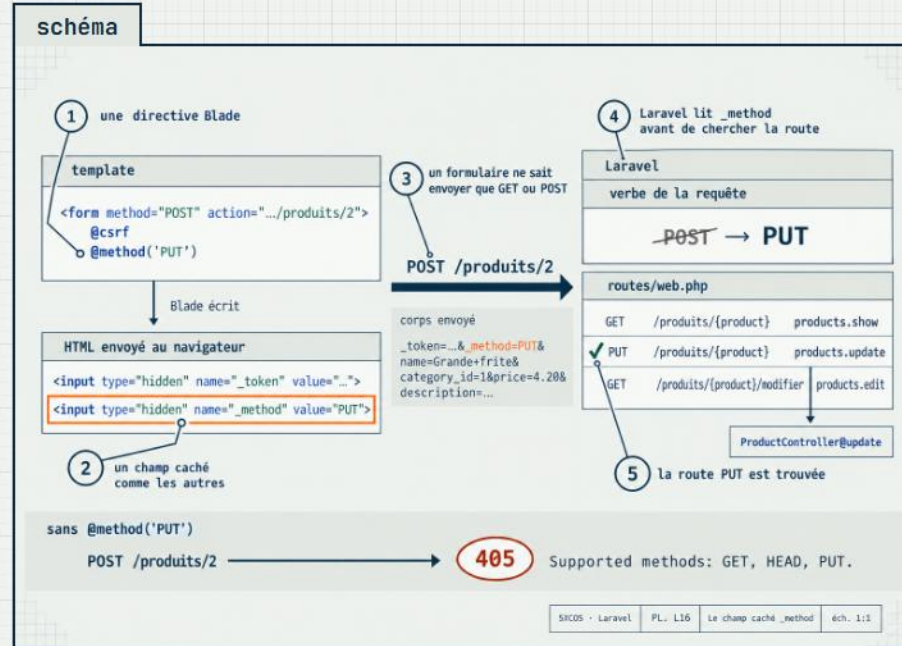
Affichez le code source de la page avec Ctrl+U. @csrf et @method('PUT') ont écrit deux champs cachés :

navigateur · code source de /produits/2/modifier

```
<form method="POST" action="http://friterie-26.test/produits/2" class="...">
  <input type="hidden" name="_token" value="K5vj..." autocomplete="off">
```

```
<input type="hidden" name="_method" value="PUT">
```

Le formulaire part en POST. Avant de chercher la route, Laravel regarde si une requête POST contient un champ _method. Si oui, il prend sa valeur comme verbe.



update() valide les champs, puis modifie la ligne

La méthode update() reçoit la requête et le produit. Elle valide les champs comme store() :

app/Http/Controllers/ProductController.php

```
1 | public function update(Request $request, Product
   | $product): RedirectResponse
2 | {
3 |     $validated = $request->validate([
4 |         'name' => ['required', 'string', 'max:60'],
   |         ...
6 |         'description' => ['nullable', 'string',
   | 'max:500'],
7 |     ]);
8 |
9 |     $product->update($validated);
10 |
11 |     return redirect()->route('products.show',
   | $product->id)->with('status', 'Le produit a été
   | modifié.');
```

navigateur · /produits/2

Le produit a été modifié.
4,20 €

Tapez 4.20 dans le prix de la Grande frite et enregistrez. Le formulaire envoie tous les champs, et update() les remplace tous : PUT remplace toute la ressource.

Sans @method(' PUT '), Laravel répond 405

Retirez la ligne @method(' PUT ') et envoyez de nouveau le formulaire :

navigateur · /produits/2

! Method Not Allowed

Copy as Markdown

Symfony\Component\HttpKernel\Exception\MethodNotAllowedHttpException

vendor\laravel\framework\src\Illuminate\Routing\AbstractRouteCollection.php:130

The POST method is not supported for route produits/2. Supported methods: GET, HEAD, PUT.

LARAVEL 13.35.0 PHP 8.4.12

UNHANDLED

CODE 0

NON CONFORME

L'adresse /produits/2 n'a pas de route POST : elle n'accepte que GET, HEAD et PUT. Remettez la ligne.

make:request range les règles d'un formulaire dans une classe à part

store() et update() répètent les mêmes quatre règles. php artisan make:request ProductRequest crée une classe de ce genre, une **Form Request** : elle contient les règles d'un formulaire.

app/Http/Requests/ProductRequest.php

```
1 | class ProductRequest extends
   | FormRequest
2 | {
   |
   | ...
4 |     public function rules(): array
5 |     {
6 |         return [
7 |             //
8 |         ];
9 |     }
10| }
```

app/Http/Controllers/ProductController.php

```
1 | public function update(① ProductRequest $request,
   | Product $product): RedirectResponse
2 | {
3 |     $product->update($request->validated());
4 |
5 |     return redirect()->route('products.show',
   | $product->id)->with('status', 'Le produit a été
   | modifié.');
```

① Le type du paramètre : Request devient ProductRequest.

Copiez le tableau de validate() dans le return de rules(), puis changez le type de \$request. \$request->validated() donne le tableau des champs validés.

La Form Request appelle `authorize()`, puis valide, avant `update()`

Enregistrez la Grande frite : Laravel répond par le code 403, « interdit ».

navigateur · /produits/2

403 | Cette action n'est pas autorisée.

app/Http/Requests/ProductRequest.php

```
1 public function authorize(): bool
2 {
3     return false;
4 }
```

`authorize()` vérifie le droit d'envoyer ce formulaire. Le site de Nadia n'a pas de comptes : faites-lui renvoyer `true`.

Une règle de validation s'écrit une fois, dans la Form Request, et sert à chaque méthode qui la reçoit.

storeProduct() crée un produit dans la catégorie de l'adresse

Sur /categories/3, la page des sauces, un formulaire court suffirait : la catégorie est déjà dans l'adresse.

routes/web.php

```
1 | Route::get(① '/categories/{category}', [CategoryController::class, 'show'])->name('categories.show');
2 | Route::post('/categories/{category}/produits', [CategoryController::class,
    'storeProduct'])->name('categories.products.store');
```

① {id} devient {category} : la liaison de modèle.

app/Http/Controllers/CategoryController.php

```
1 | public function storeProduct(ProductRequest $request, Category $category): RedirectResponse
2 | {
3 |     $category->products()->create($request->validated());
4 |
5 |     return redirect()->route('categories.show', $category->id)->with('status', 'Le produit a été ajouté.');
```

products(), avec parenthèses, renvoie la relation du chapitre 11. Sa méthode create() enregistre un produit et remplit elle-même category_id avec l'id de la catégorie.

Le formulaire court n'envoie que le nom et le prix

Dans `categories/show.blade.php`, sous la liste des produits, ajoutez le formulaire, sans `<select>` :

resources/views/categories/show.blade.php

```
1 | <form method="POST" action="{{ route('categories.products.store', $category->id) }}" class="mt-12
   | max-w-2xl 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">
2 |     @csrf
3 |
4 |     <h2 class="text-xl font-semibold tracking-tight">Ajouter un produit dans cette catégorie</h2>
   |
   | ...
8 |     <x-input name="name" :value="old('name')" />
   |
   | ...
16 |    <x-input name="price" type="number" step="0.10" min="0" :value="old('price')" />
   |
   | ...
23 |    <x-button type="submit">Ajouter à {{ $category->name }}</x-button>
   |
   | ...
25 | </form>
```

Tapez Sauce cocktail et 0.80, puis envoyez. Le formulaire revient avec les deux valeurs, sans message et sans encadré vert : rien n'a été enregistré.

prepareForValidation() ajoute la catégorie de l'adresse avant la validation

ProductRequest exige category_id, que le formulaire court n'envoie pas. Avant de valider, Laravel appelle la méthode prepareForValidation() de la Form Request, si elle existe.

app/Http/Requests/ProductRequest.php

```
1 | protected function prepareForValidation():  
   | void  
2 | {  
3 |     if (① $this->route('category')) {  
② 4 |         $this->merge(['category_id' =>  
   |             $this->route('category')->id]);  
5 |     }  
6 | }
```

- ① La catégorie de l'adresse, ou null sans {category}.
- ② Ajoute category_id aux champs de la requête.

Avec dd(\$request->validated()); en tête de storeProduct(), category_id vaut le nombre 3, sans guillemets : il vient de l'adresse.

navigateur · /categories/3/produits

```
array:3 [▼ //  
  app/Http/Controllers/CategoryController.php:24  
  "name" => "Sauce cocktail"  
  "category_id" => 3  
  "price" => "0.80"  
]
```

edit.blade.php se réduit à l'appel du composant

<x-product-fields>

Nadia doit cocher les allergènes, à l'ajout comme à la modification. `php artisan make:component product-fields --view` crée un composant anonyme, qui reçoit les quatre champs et les cases.

resources/views/products/edit.blade.php

```
1 | <form method="POST" action="{ route('products.update',  
    $product->id) }" class="...">  
2 |     @csrf  
3 |     @method('PUT')  
4 |  
5 |     <x-product-fields ① :product="$product"  
        :categories="$categories" :allergens="$allergens" />  
6 |  
7 |     <div class="flex justify-end">  
8 |         <x-button type="submit">Enregistrer</x-button>  
9 |     </div>  
10| </form>
```

① Les deux-points transmettent la variable, pas son nom.

`new Product` crée un objet absent de la base : ses colonnes valent `null`, et il n'a aucun allergène. `create.blade.php` appelle le même composant.

app/Http/Controllers/ProductController.php · create()

```
1 | return view('products.create', [  
2 |     'product' => new Product,  
3 |     'categories' => $categories,  
4 |     'allergens' => Allergen::all(),  
5 | ]);
```

Chaque case allergens[] porte l'id de son allergène

Sous la description, le composant ajoute une case par allergène :

resources/views/components/product-fields.blade.php

```
46 <fieldset>
47     <legend class="text-sm font-medium">Allergènes</legend>
48     <div class="mt-2 grid grid-cols-2 gap-2 sm:grid-cols-4">
49         @foreach ($allergens as $allergen)
50             <label class="flex items-center gap-2 text-sm">
51                 <input type="checkbox" ① name="allergens[]" value="{{ $allergen->id }}"
52                 ② @checked(in_array($allergen->id, $checkedAllergens)) class="size-4 accent-brand-600">
53                     {{ $allergen->name }}
54             </label>
55         @endforeach
56     </div>
57     ...
58 </fieldset>
```

① Les crochets : PHP range les valeurs cochées dans un tableau.

② Écrit checked si l'id est dans \$checkedAllergens.

Dans le code source de /produits/3/modifier, la case Gluten de la fricadelle est cochée :

navigateur · code source de /produits/3/modifier

```
<input type="checkbox" name="allergens[]" value="1" checked class="size-4 accent-brand-600">
```

Un navigateur n'envoie pas une case décochée

Ajoutez `dd($request->all());` en tête d'`update()`. Cochez Gluten et Soja pour la viande et enregistrez, puis décochez-les et enregistrez de nouveau.

navigateur · cases cochées

```
array:7 [▼ // app/Http/Controllers/ProductController.php:67
  ""
  "description" => "Une saucisse de viande hachée, frite à la commande."
  "allergens" => array:2 [▼
    0 => "1"
    1 => "5"
  ]
]
```

navigateur · cases décochées

```
array:6 [▼ // app/Http/Controllers/ProductController.php:67
  ""
  "description" => "Une saucisse de viande hachée, frite à la commande."
]
```

Sans case cochée, la clé `allergens` est absente de la requête : le tableau passe de sept à six clés.

session()->hasOldInput() dit si l'on revient d'un refus

Après un refus, une liste absente veut dire qu'aucune case n'était cochée. Au premier affichage, la session ne contient rien, et les cases reprennent les allergènes enregistrés.

resources/views/components/product-fields.blade.php

```

① 1 | @props(['product', 'categories', 'allergens'])
   2 |
   3 | @php
   4 |     $checkedAllergens = ② session()->hasOldInput()
   5 |         ? (array) old('allergens', [])
   6 |         : $product->allergens->pluck('id')->all();
   7 | @endphp

```

- ① Ce que le composant attend, comme au chapitre 9.
- ② Un envoi précédent a laissé des valeurs en session ?
- ③ Sinon, les id des allergènes du produit.

Sur la fricadelle, décochez Gluten et Soja, tapez un prix de 0, puis enregistrez. Après le refus, la case Gluten reste décochée :

navigateur · code source de /produits/3/modifier

```
<input type="checkbox" name="allergens[]" value="1" class="size-4 accent-brand-600">
```

(array) protège la vue d'une valeur seule au lieu d'un tableau

Avec curl, ou en renommant la case dans les outils de développement, on envoie `allergens=1`, une valeur seule. `old('allergens', [])` renvoie alors la chaîne "1", et `in_array()` s'arrête :

navigateur · page d'erreur

```
in_array(): Argument #2 ($haystack) must be of type array, string given
```

(array) convertit cette chaîne en tableau d'un élément, ["1"]. **Une vue ne fait pas plus confiance à la requête qu'un contrôleur.**



allergens.* applique ses règles à chaque case reçue

Ajoutez deux règles dans ProductRequest. Dans allergens.*, l'étoile désigne chaque élément du tableau :

app/Http/Requests/ProductRequest.php · rules()

```
① 1 | 'allergens' => ['nullable', 'array'],  
   2 | 'allergens.*' => ['integer',  
   ② | 'exists:allergens,id'],
```

- ① Un tableau, ou rien.
- ② Chaque valeur : l'id d'une ligne d'allergens.

Une case trafiquée à 99 donne « Le champ allergens.1 sélectionné est invalide. ». attributes() nomme le champ : « Le champ allergène sélectionné est invalide. »

app/Http/Requests/ProductRequest.php

```
1 | public function attributes(): array  
2 | {  
3 |     return [  
4 |         'allergens.*' => 'allergène',  
5 |     ];  
6 | }
```

`sync($validated['allergens'] ?? [])` remplace la liste entière, vide comprise

Dans `update()`, enregistrez la liste cochée. Faites le même ajout dans `store()`.

app/Http/Controllers/ProductController.php · `update()`

```
1 | $validated = $request->validated();  
2 |  
3 | $product->update($validated);  
4 | $product->allergens()->sync($validated['allergens'] ?? []);
```

`?? []` donne un tableau vide quand aucune case n'est cochée, et `sync([])` retire alors tous les allergènes. La table pivot de la viande, avant puis après Gluten et Soja :

terminal

```
$ php artisan tinker --execute="echo DB::table('allergen_product')->where('product_id', 17)->get();"
[]  
$ php artisan tinker --execute="echo DB::table('allergen_product')->where('product_id', 17)->get();"
[{"allergen_id":1,"product_id":17},{ "allergen_id":5,"product_id":17}]
```

La rupture ne change qu'une colonne : une route PATCH

Un samedi soir, les croquettes manquent. Le badge « En rupture » s'affiche dès que `sold_out_at` contient une date. PATCH ne change qu'une partie de la ressource.

routes/web.php

```
1 | Route::patch('/produits/{product}/rupture', [ProductController::class,  
    'soldOut'])->name('products.sold-out');
```

app/Http/Controllers/ProductController.php

```
1 | public function soldOut(Product $product): RedirectResponse  
2 | {  
3 |     if (! $product->isSoldOut()) {  
4 |         $product->update(['sold_out_at' => now()]);  
5 |     }  
6 |  
7 |     return redirect()->route('products.show', $product->id)->with('status', 'Le produit est marqué  
    en rupture.');
```

```
8 | }
```

`now()` renvoie la date et l'heure actuelles. Le `if` garde la date de la première rupture : un second envoi ne la change pas.

Un formulaire d'un seul bouton marque le produit en rupture

Sur la fiche, ajoutez un formulaire affiché seulement pour un produit disponible :

resources/views/products/show.blade.php

```
1 | @if (! $product->isSoldOut())
2 |     <form method="POST" action="{{
    route('products.sold-out', $product->id) }}">
3 |         @csrf
4 |         @method('PATCH')
5 |         <x-button type="submit"
    variant="secondary">Marquer en rupture</x-button>
6 |     </form>
7 | @endif
```

navigateur · /produits/5, après le clic

Le produit est marqué en rupture.
Croquette de fromage
En rupture
En rupture depuis 07/10/2026, 15 h 22

L'adresse désigne le produit, et le verbe PATCH dit quoi en faire. Un lien aurait été plus court, mais un robot qui le suit marquerait le produit en rupture.

Route::delete() et destroy() retirent un produit de la carte

Nadia ne vend plus de sauce tartare. Pour supprimer, HTTP propose le verbe DELETE. La route reprend l'adresse d'update(), avec un autre verbe :

routes/web.php

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

app/Http/Controllers/ProductController.php

```
1 | public function destroy(Product $product): RedirectResponse  
2 | {  
3 |     $product->delete();  
4 |  
5 |     return redirect()->route('products.index')->with('status', 'Le produit a été retiré  
   | de la carte.');
```

```
6 | }
```

La fiche n'existe plus après la suppression : la redirection mène à la carte. Les lignes du produit dans `allergen_product` disparaissent aussi, grâce au `cascadeOnDelete()` de la migration.

confirm() demande une confirmation avant d'envoyer la suppression

Sur la fiche, après le formulaire de rupture, ajoutez :

resources/views/products/show.blade.php

```
1 | <form method="POST" action="{{ route('products.destroy', $product->id) }}"
   |   ① onsubmit="return ② confirm('Retirer ce produit de la carte ?')">
2 |     @csrf
3 |     @method('DELETE')
4 |     <x-button type="submit" variant="danger">Retirer de la carte</x-button>
5 | </form>
```

- ① Du JavaScript, exécuté juste avant l'envoi.
- ② Une boîte du navigateur : OK ou Annuler.

confirm() renvoie false sur Annuler, et return false arrête l'envoi. Il évite un clic malheureux, mais ne protège rien côté serveur : une requête envoyée sans le formulaire ne passe par aucune boîte.

route:list montre les routes des produits, écrites à la main

terminal

```
$ php artisan route:list --path=produits
POST      categories/{category}/produits ..... categories.products.store > CategoryController@storeProduct
POST      produits ..... products.store > ProductController@store
GET|HEAD  produits/nouveau ..... products.create > ProductController@create
GET|HEAD  produits/{product} ..... products.show > ProductController@show
PUT       produits/{product} ..... products.update > ProductController@update
DELETE    produits/{product} ..... products.destroy > ProductController@destroy
GET|HEAD  produits/{product}/modifier ..... products.edit > ProductController@edit
PATCH    produits/{product}/rupture ..... products.sold-out > ProductController@soldOut
```

Showing [8] routes

Avec `products.index` sur `/carte`, `ProductController` donne un corps aux sept méthodes que `RecipeController --resource` laissait vides au chapitre 8 : `index`, `create`, `store`, `show`, `edit`, `update` et `destroy`.

Route::resource() déclare sept routes, mais pas les vôtres

Ajoutez temporairement cette ligne à la fin de routes/web.php, puis listez les routes qu'elle déclare :

routes/web.php

```
1 | Route::resource('produits', ProductController::class);
```

terminal

```
$ php artisan route:list --name=produits
POST          produits ..... produits.store > ProductController@store
GET|HEAD      produits ..... produits.index > ProductController@index
GET|HEAD      produits/create ..... produits.create > ProductController@create
GET|HEAD      produits/{produit} ..... produits.show > ProductController@show
PUT|PATCH    produits/{produit} ..... produits.update > ProductController@update
DELETE        produits/{produit} ..... produits.destroy > ProductController@destroy
GET|HEAD      produits/{produit}/edit ..... produits.edit > ProductController@edit
```

Showing [7] routes

Les adresses finissent par `create` et `edit`, en anglais, et les noms commencent par `produits..` Le paramètre `{produit}` ne porte pas le nom de l'argument `$product` : pas de liaison de modèle. Le cours garde ses routes écrites à la main.

Exercice 16.1, sur papier : qui reçoit chacune de ces requêtes ?

Relisez le route:List de la section 8. Pour chaque requête, écrivez comme le navigateur l'envoi, dites quelle méthode de quel contrôleur la reçoit. Si aucune ne la reçoit, donnez le code de la réponse.

16.1 • sur papier, 15 minutes

1. GET /produits/4/modifier
2. POST /produits/4 avec _method=PUT
3. POST /produits/4 avec _method=PATCH
4. POST /produits/4/rupture avec _method=PATCH
5. GET /produits/4/rupture
6. POST /categories/2/produits
7. GET /produits/modifier

Les katas du refuge, au chapitre 17, viennent ensuite : ils reprennent tout ce que les chapitres 15 et 16 ont montré sur la friterie.