TP6 — Professionnaliser un CRUD avec Form Requests, recherche et pagination

1. Objectifs du TP

À la fin de ce TP, vous serez capables de :

- comprendre pourquoi un contrôleur peut devenir trop chargé ;
- créer un **Form Request** Laravel ;
- déplacer les règles de validation dans une classe dédiée ;
- utiliser l'injection de dépendances ;
- simplifier un contrôleur ;
- rechercher des clients dans la base de données ;
- utiliser les paramètres d'une URL ;
- afficher les résultats avec une pagination Laravel.

2. Situation de départ

À la fin du TP5, notre `ClientController` contient désormais la validation.

Par exemple :

```php
public function store(Request $request)
{
    $request->validate([
        'name' => 'required|string|max:100',
        'email' => 'required|email',
        'phone' => 'nullable|string|max:20',
        'city' => 'nullable|string|max:100',
    ]);

    Client::create($request->all());

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

Et dans `update()` nous avons pratiquement les mêmes règles :

```php
public function update(Request $request, Client $client)
{
    $request->validate([
        'name' => 'required|string|max:100',
```

```
            'email' => 'required|email',
            'phone' => 'nullable|string|max:20',
            'city' => 'nullable|string|max:100',
        ]);

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

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

Notre application fonctionne. Mais nous avons maintenant un nouveau problème : **La validation est mélangée avec la logique du contrôleur.**

Et surtout, nous avons écrit pratiquement deux fois les mêmes règles. Nous allons améliorer notre architecture.

3. Exercice 1 / Identifier le problème

Observez votre `ClientController`.

Il contient maintenant plusieurs responsabilités :

```
ClientController
│
├── reçoit la requête
├── valide les données
├── récupère les clients
├── crée un client
├── modifie un client
├── supprime un client
└── redirige l'utilisateur
```

Question

Est-ce que toutes ces responsabilités devraient obligatoirement être écrites directement dans le contrôleur ?

Dans une petite application, cela peut fonctionner.

Mais dans une application plus importante, les contrôleurs peuvent rapidement devenir très longs et difficiles à maintenir.

Nous allons donc commencer à **séparer les responsabilités**.

4. Les Form Requests

Laravel propose une classe appelée :

Form Request

Un Form Request permet notamment de regrouper :

- les règles de validation ;
- l'autorisation de la requête.

Nous allons créer un Form Request pour la création d'un client.

5. Exercice 2 / Créer StoreClientRequest

Dans le terminal, exécutez :

```
php artisan make:request StoreClientRequest
```

Laravel crée :

```
app/
└── Http/
    └── Requests/
        └── StoreClientRequest.php
```

Ouvrez le fichier. Vous trouverez notamment :

```php
class StoreClientRequest extends FormRequest
{
    public function authorize(): bool
    {
        return false;
    }

    public function rules(): array
    {
        return [
        ];
    }
}
```

6. Comprendre `authorize()`

Pour l'instant, nous allons écrire :

```php
public function authorize(): bool
{
    return true;
}
```

Pourquoi ?

Parce que :

```php
return true;
```

signifie que cette requête est autorisée.

Nous étudierons plus précisément les autorisations et les utilisateurs lors du chapitre consacré à **l'authentification et à la sécurité**.

Pour le moment :

```php
public function authorize(): bool
{
    return true;
}
```

7. Exercice 3 / Déplacer les règles de validation

Nous allons maintenant déplacer les règles que nous avions écrites dans `ClientController`.

Dans `StoreClientRequest.php` :

```php
public function rules(): array
{
    return [
        'name' => 'required|string|max:100',
        'email' => 'required|email',
        'phone' => 'nullable|string|max:20',
        'city' => 'nullable|string|max:100',
    ];
}
```

La classe complète doit maintenant ressembler à :

```php
<?php

namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest;

class StoreClientRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            'name' => 'required|string|max:100',
            'email' => 'required|email',
            'phone' => 'nullable|string|max:20',
            'city' => 'nullable|string|max:100',
        ];
    }
}
```

8. Exercice 4 / Utiliser le Form Request dans le contrôleur

Dans `ClientController.php`, nous avions :

```php
use Illuminate\Http\Request;
```

et :

```php
public function store(Request $request)
```

Nous allons maintenant importer notre Form Request :

```php
use App\Http\Requests\StoreClientRequest;
```

Puis modifier la méthode :

```php
public function store(StoreClientRequest $request)
{
    Client::create($request->validated());

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

Observez la différence

Avant :

```
public function store(Request $request)
{
    $request->validate([
        ...
    ]);

    Client::create($request->all());
}
```

Après :

```
public function store(StoreClientRequest $request)
{
    Client::create($request->validated());
}
```

Le contrôleur est beaucoup plus simple.

9. Utilisation de `validated()` plutôt que `all()`

Dans le TP5, nous utilisions :

```
$request->all()
```

Cette méthode récupère toutes les données envoyées dans la requête.

Avec un Form Request, nous pouvons utiliser :

```
$request->validated()
```

Cette méthode retourne uniquement les données qui ont été validées.

À retenir

```
$request->all()            >>>      Toutes les données reçues
$request->validated()      >>>      Les données validées
```

Dans notre application, nous préférons donc :

```
Client::create($request->validated());
```

10. Test

Testez à nouveau la création d'un client.

Test valide

```
Nom : Jean Dupont
Email : jean@test.fr
Téléphone : 0601020304
Ville : Paris
```

Le client doit être créé.

Test invalide

```
Nom :
Email : bonjour
```

La validation doit toujours fonctionner.

Question

Avons-nous changé le fonctionnement du formulaire ? **Non.**

Nous avons simplement changé **l'organisation de notre code**.

11. Exercice 5 / Créer UpdateClientRequest

Nous allons maintenant faire la même chose pour la modification.

Exécutez :

```
php artisan make:request UpdateClientRequest
```

Laravel crée :

```
app/Http/Requests/UpdateClientRequest.php
```

Modifiez `authorize()` :

```php
public function authorize(): bool
{
    return true;
}
```

Puis ajoutez les mêmes règles :

```php
public function rules(): array
{
    return [
        'name' => 'required|string|max:100',
        'email' => 'required|email',
        'phone' => 'nullable|string|max:20',
        'city' => 'nullable|string|max:100',
    ];
}
```

12. Exercice 6 / Nettoyer `update()`

Importez :

```php
use App\Http\Requests\UpdateClientRequest;
```

Puis modifiez :

```php
public function update(UpdateClientRequest $request, Client $client)
{
    $client->update($request->validated());

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

Nous avons maintenant :

```
ClientController
        │
        ├── StoreClientRequest
        │          ↓
        │     Validation
        │
        └── UpdateClientRequest
                   ↓
              Validation
```

Le contrôleur n’a plus besoin de connaître le détail des règles de validation.

13. Exercice 7 / Comprendre l’injection de dépendances

Observez cette méthode :

```
public function update(
    UpdateClientRequest $request,
    Client $client
)
```

Laravel va automatiquement fournir les objets nécessaires à la méthode.

C’est ce qu’on appelle **l’injection de dépendances**.

Le principe est simplement :

> Au lieu de créer nous-mêmes les objets dont une méthode a besoin, nous les déclarons dans ses paramètres et Laravel peut les fournir.

Par exemple :

```
public function store(StoreClientRequest $request)
```

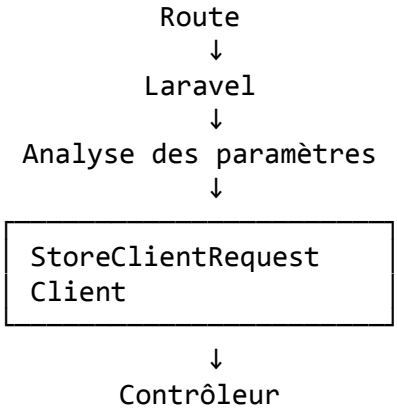
Laravel comprend que la méthode attend un objet `StoreClientRequest`.

De même :

```
public function show(Client $client)
```

Laravel récupère automatiquement le client correspondant à l’identifiant présent dans l’URL grâce au **route model binding** étudié au TP4.

Schéma



Pour l’instant, retenez simplement le principe :

On déclare ce dont on a besoin ; Laravel s’occupe de nous le fournir.

14. Exercice 8 / Ajouter une recherche de clients

Notre Mini-CRM contient maintenant plusieurs clients. Mais imaginez que nous ayons 500 clients. Afficher toute la liste devient peu pratique.

Nous allons ajouter une recherche.

Nous voulons pouvoir utiliser une URL comme :

`/clients?search=jean`

Le paramètre :

`search=jean`

est un **paramètre de requête HTTP**.

15. Modifier `index()`

Ouvrez : `app/Http/Controllers/ClientController.php`

Nous avions probablement :

```php
public function index()
{
    $clients = Client::all();
    return view('clients.index', compact('clients'));
}
```

Nous allons récupérer le paramètre `search`.

```php
public function index(Request $request)
{
    $search = $request->input('search');
    $clients = Client::where('name', 'like', '%' . $search . '%')
                     ->get();
    return view('clients.index', compact('clients', 'search'));
}
```

16. Comprendre la requête

Cette partie :

```
Client::where('name', 'like', '%' . $search . '%')
```

signifie :

> Chercher les clients dont le nom contient la valeur recherchée.

Si :

```
search = jean
```

Laravel va construire une requête équivalente à :

```
SELECT *
FROM clients
WHERE name LIKE '%jean%';
```

Ainsi :

```
Jean Dupont
Jean Martin
Pierre Jean
```

peuvent être trouvés.

17. Attention au cas où aucune recherche n’est effectuée

Si l’utilisateur visite simplement : `/clients`

nous ne voulons pas forcément effectuer une recherche avec : `%%`

Nous allons donc utiliser une condition.

Une première solution :

```
public function index(Request $request)
{
    $search = $request->input('search');

    $query = Client::query();

    if ($search) {
        $query->where('name', 'like', '%' . $search . '%');
```

```
    }

    $clients = $query->get();

    return view('clients.index', compact('clients', 'search'));
}
```

Le principe

```
/clients                    >>>      Tous les clients

/clients?search=jean        >>>      Clients dont le nom contient "jean"
```

18. Exercice 9 / Ajouter le formulaire de recherche

Dans :

`resources/views/clients/index.blade.php`

Ajoutez :

```
<form method="GET" action="{{ route('clients.index') }}">
    <input
        type="text"
        name="search"
        placeholder="Rechercher un client"
        value="{{ $search }}"
    >

    <button type="submit">
        Rechercher
    </button>
</form>
```

Pourquoi GET ?

Parce que la recherche ne modifie pas les données.

Nous demandons simplement :

> « Donne-moi les clients correspondant à ce critère. »

Le résultat est donc naturellement représenté dans l'URL :

`/clients?search=jean`

19. Exercice 10 / Améliorer la recherche

Pour le moment, nous recherchons uniquement dans le nom.

Nous allons rechercher également dans :

- le nom ;
- l'e-mail ;
- la ville.

Utilisez :

```
if ($search) {
    $query->where(function ($query) use ($search) {
        $query->where('name', 'like', '%' . $search . '%')
              ->orWhere('email', 'like', '%' . $search . '%')
              ->orWhere('city', 'like', '%' . $search . '%');
    });
}
```

Nous pouvons maintenant rechercher :

```
/clients?search=paris
```

ou :

```
/clients?search=gmail
```

ou :

```
/clients?search=jean
```

20. Exercice 11 / Ajouter la pagination

Notre application fonctionne maintenant. Mais imaginons que nous ayons 10 000 clients. En afficher autant sur une seule page serait une mauvaise idée.

Nous allons donc afficher seulement **10 clients par page**.

Remplacez :

```
$clients = $query->get();
```

par :

```
$clients = $query->paginate(10);
```

Laravel va automatiquement gérer la pagination.

21. Afficher les liens de pagination

Dans :

`resources/views/clients/index.blade.php`

Ajoutez après le tableau :

```
{{ $clients->links() }}
```

Laravel va générer les liens permettant de naviguer entre les pages.

Par exemple :

```
← Previous    1    2    3    4    Next →
```

22. Conserver la recherche lors du changement de page

Il reste un petit problème.

Si nous sommes sur :

`/clients?search=jean`

et que nous cliquons sur la page 2, nous voulons conserver :

`search=jean`

Modifiez donc :

```
$clients = $query->paginate(10);
```

en :

```
$clients = $query
    ->paginate(10)
    ->withQueryString();
```

Ainsi, Laravel conservera les paramètres de l'URL lors de la navigation.

23. Exercice 12 / Finaliser `index()`

Votre méthode `index()` doit maintenant ressembler à :

```
public function index(Request $request)
{
    $search = $request->input('search');

    $query = Client::query();

    if ($search) {
        $query->where(function ($query) use ($search) {
            $query->where('name', 'like', '%' . $search . '%')
                  ->orWhere('email', 'like', '%' . $search . '%')
                  ->orWhere('city', 'like', '%' . $search . '%');
        });
    }

    $clients = $query
        ->paginate(10)
        ->withQueryString();

    return view('clients.index', compact('clients', 'search'));
}
```

24. Test complet

Ajoutez suffisamment de clients dans votre base SQLite pour avoir plus de 10 clients.

Vous pouvez utiliser votre Seeder.

Puis testez :

Test 1

`/clients`

Résultat : Tous les clients sont affichés, 10 par page.

Test 2

`/clients?search=jean`

Résultat : Seuls les clients correspondant à la recherche sont affichés.

Test 3

Passez à la page 2.
Vérifiez que la recherche `jean` est toujours présente.

25. Exercice final / Professionnaliser le CRUD Produit

Nous allons maintenant appliquer toutes les notions du TP à notre deuxième ressource : **Product**.

Votre CRUD Product existe déjà depuis le TP4.

Vous devez maintenant le professionnaliser.

Étape 1 / Form Requests

Créer :

```
php artisan make:request StoreProductRequest
```

et :

```
php artisan make:request UpdateProductRequest
```

Utilisez les règles :

```
'name' => 'required|string|max:150',
'price' => 'required|numeric|min:0',
'stock' => 'required|integer|min:0',
```

Étape 2 / Nettoyer le contrôleur

Le `ProductController` doit utiliser :

`StoreProductRequest`

pour `store()` et :

`UpdateProductRequest`

pour `update()`.

Les méthodes doivent utiliser :

```
$request->validated()
```

et non :

```
$request->all()
```

Étape 3 / Ajouter la recherche

La page :

`/products`

doit permettre de rechercher un produit :

`/products?search=clavier`

La recherche doit porter sur :

`name`

Étape 4 / Ajouter la pagination

Afficher :

`10 produits par page`

et ajouter :

```
{{ $products->links() }}
```

26. Challenge bonus

Si vous avez terminé avant la fin du TP, ajoutez une recherche sur plusieurs champs pour les produits.

Par exemple :

`/products?search=15`

Vous pouvez réfléchir à la possibilité de rechercher :

- dans le nom ;
- dans le prix.

Question

Comment modifieriez-vous votre requête ?

27. Bilan du TP

Au début du TP, notre contrôleur contenait directement la validation :

```
Controller
    ↓
Validation
    ↓
Eloquent
```

Nous avons séparé les responsabilités :

```
Request
   ↓
Form Request
   ↓
Validation
   ↓
Controller
   ↓
Eloquent
   ↓
SQLite
```

Nous avons également appris à exploiter les paramètres d’une URL :

```
/clients?search=jean
```

et à utiliser la pagination :

```
Client::paginate(10);
```

28. Les notions à retenir

Form Request

Créer une classe dédiée à la validation :

```
php artisan make:request StoreClientRequest
```

Règles de validation

```
public function rules(): array
{
    return [
        'name' => 'required|string|max:100',
        'email' => 'required|email',
    ];
}
```

Utilisation dans le contrôleur

```
public function store(StoreClientRequest $request)
{
    Client::create($request->validated());
}
```

Injection de dépendances

```
public function store(StoreClientRequest $request)
```

Laravel fournit automatiquement l'objet nécessaire.

Recherche

```
$request->input('search');
```

Requête dynamique

```
$query = Client::query();

if ($search) {
    $query->where(...);
}
```

Pagination

```
Client::paginate(10);
```

Liens de pagination

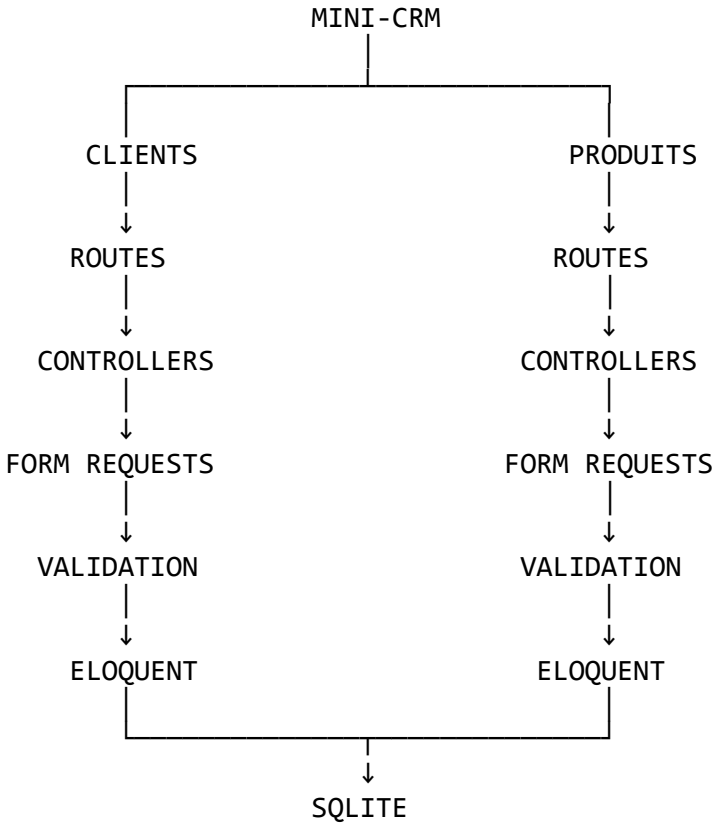
```
{{ $clients->links() }}
```

Conservation des paramètres

```
->withQueryString();
```

Architecture obtenue après le TP6

Notre Mini-CRM commence maintenant à avoir une architecture propre :



À ce stade, nous avons donc appris :

```
TP1 / TP2
Routes / Controllers / Blade
        ↓
TP3 / TP4
SQLite / DBeaver / Eloquent / CRUD
        ↓
TP5
Validation des formulaires
        ↓
TP6
Form Requests / Injection / Recherche / Pagination
```

Prochaine étape : nous pourrons aborder l'**authentification et la gestion des utilisateurs**, ce qui permettra de faire évoluer le Mini-CRM d'une simple application CRUD vers une véritable application web avec des utilisateurs et des droits d'accès.

TP6 - Sujet complété

Réponses et résultats des exercices

Les 20 pages originales sont conservées avant cette annexe. Le PDF fourni n'a aucun champ interactif ; les réponses sont ajoutées ici dans l'ordre du sujet.

Pages 1-2 | Objectif et question sur le contrôleur

Réponse : non, toutes les responsabilités n'ont pas besoin d'être écrites dans le contrôleur. Il coordonne la requête et la réponse ; un Form Request regroupe l'autorisation, les règles et les messages de validation. Cela rend store() et update() plus courts et permet de retrouver les règles dans un endroit prévu pour elles.

Point de départ réel

Le sujet montre Client::create($request->all()). Dans le projet après TP5, les contrôleurs utilisaient déjà une méthode validatedData() et ne transmettaient que des valeurs validées. Le TP6 déplace ces règles dans quatre classes Form Request ; le comportement de validation reste inchangé.

Pages 3-8 | Form Requests Client

Exercice	Réalisation
2-3 : création	StoreClientRequest.php : authorize() = true ; règles name, email, phone, city ; messages français du TP5.
4 : store()	store(StoreClientRequest $request) ; Client::create($request->validated()) ; redirection vers clients.index.
5 : modification	UpdateClientRequest.php : mêmes règles ; la règle unique ignore le client modifié.
6 : update()	update(UpdateClientRequest $request, Client $client) ; $client->update($request->validated()).

Pourquoi authorize() retourne true ? Le TP6 ne demande pas encore d'authentification ni de droits par utilisateur. La requête est donc autorisée à atteindre la validation. Ce choix ne signifie pas qu'un contrôle d'accès n'aurait jamais d'intérêt.

all() contre validated() : all() récupère l'ensemble des entrées reçues ; validated() ne renvoie que les champs acceptés par les règles. Le contrôleur utilise validated() pour l'écriture Eloquent.

Test page 7 : la création valide fonctionne toujours ; un nom vide et un email invalide sont toujours refusés. Le formulaire se comporte comme au TP5. Seule l'organisation du code a changé.

Page 9 | Injection de dépendances

Laravel fournit au contrôleur les objets déclarés dans ses paramètres : StoreClientRequest pour la création, UpdateClientRequest et Client pour la modification. Le Client provient de l'identifiant de l'URL via le route model binding. Le Form Request est validé avant l'exécution du corps de store() ou update().

Pages 10-12 | Recherche Client

Le formulaire GET transmet search dans l'URL, par exemple /clients?search=jean. index(Request $request) récupère cette valeur, construit Client::query() et n'ajoute une condition que si la recherche n'est pas vide. Sans recherche, tous les clients sont admissibles.

GET est choisi parce que chercher ne crée, ne modifie et ne supprime aucun enregistrement. L'URL peut être copiée ou enregistrée dans les favoris.

Page 13 | Nom, email et ville

Les trois critères sont groupés dans un même where(function ...). Le nom, l'email et la ville sont reliés par OR **à l'intérieur du groupe**. Ainsi, « Paris » peut trouver un client par ville et « gmail » par email. Les valeurs sont passées au constructeur de requête Laravel, sans concaténer une requête SQL brute.

Pages 13-15 | Pagination

paginate(10) affiche au plus dix lignes par page. links() génère les liens. withQueryString() conserve search quand on passe de la page 1 à la page 2. orderBy('id') stabilise l'ordre. Le total de la liste utilise total() : count() ne compterait que la page courante.

Essai du sujet	Résultat vérifié
/clients	Tous les clients correspondent ; dix au plus sont affichés par page.
/clients?search=Jean	Seuls les clients dont nom, email ou ville contient Jean sont retenus.
Page 2 après recherche	Le filtre search reste dans les liens et seuls les résultats suivants apparaissent.

Pour tester deux pages, les tests automatisés créent douze clients dans SQLite en mémoire. La base personnelle conserve ses données réelles.

Pages 16-17 | Exercice final Produit

Étape	Réponse / réalisation
Form Requests	StoreProductRequest et UpdateProductRequest portent les règles name required|string|max:150, price required|numeric|min:0, stock required|integer|min:0.
Controller	store() et update() reçoivent les Form Requests et passent uniquement $request->validated() à Eloquent.
Recherche	GET /products?search=clavier filtre les noms contenant clavier.
Pagination	Dix produits par page ; links() dans Blade ; withQueryString() garde le filtre.

Page 17 | Challenge bonus : chercher aussi dans le prix

Réponse : regrouper deux conditions dans le même where : name LIKE %search% OR price LIKE %search%. Dans SQLite, LIKE convertit le prix numérique en texte pour cette recherche. Ainsi /products?search=15 trouve un prix 15,50. Pour un filtrage financier exact, une comparaison numérique explicite serait plus adaptée ; le défi demande une recherche de texte dans plusieurs champs.

```
$query->where(function ($query) use ($search) {
    $query->where('name', 'like', '%'.$search.'%')
          ->orWhere('price', 'like', '%'.$search.'%');
});
```

Pages 18-20 | Bilan

Le parcours est désormais : route → injection du Form Request → autorisation et validation → contrôleur → Eloquent → SQLite → redirection ou vue. Pour les listes : paramètre search → requête Eloquent groupée → paginate(10) → Blade avec links().

Vérification finale

La suite Laravel passe : **11 tests, 131 vérifications**. Elle couvre le CRUD TP4, la validation TP5, les Form Requests TP6, les recherches par nom/email/ville et nom/prix, deux pages de résultats et la conservation du filtre.