TP5 / Sécuriser et fiabiliser un CRUD avec la validation Laravel

1. Objectifs du TP

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

- récupérer les données envoyées par un formulaire ;
- comprendre pourquoi les données doivent être validées ;
- utiliser la validation Laravel ;
- définir des règles de validation ;
- afficher les erreurs dans une vue Blade ;
- conserver les données saisies après une erreur ;
- afficher un message d'erreur pour un champ précis ;
- appliquer la validation à la création et à la modification d'un client.

2. Situation de départ

Lors du TP4, nous avons créé un CRUD permettant de gérer nos clients.

Nous pouvons maintenant :

- afficher la liste des clients ;
- créer un client ;
- afficher un client ;
- modifier un client ;
- supprimer un client.

Notre méthode `store()` ressemble actuellement à ceci :

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

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

Le problème est que **nous faisons confiance aux données envoyées par l'utilisateur**.

Problème

Un formulaire HTML permet de contrôler certaines choses côté navigateur, mais **le serveur doit toujours vérifier les données reçues**.

> Une règle essentielle du développement web :
>
> **Ne jamais faire confiance aux données envoyées par l'utilisateur.**

Laravel possède un système de validation intégré qui permet de résoudre ce problème.

3. Exercice 1 / Tester le problème

Avant de modifier le code, nous allons observer le comportement actuel de notre application.

Lancez le serveur : `php artisan serve`

Puis rendez-vous sur : `/clients`

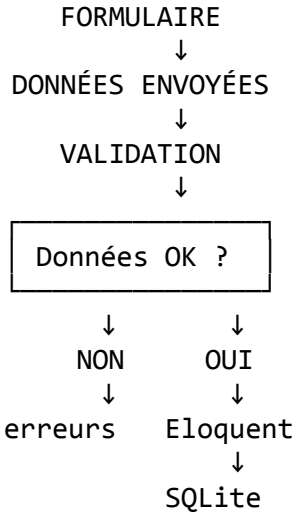
Cliquez sur **Créer un client**.

Essayez de créer un client avec :

- un nom vide ;
- une adresse e-mail incorrecte ;
- un téléphone contenant des lettres ;
- un nom extrêmement long.

Conclusion : Notre CRUD fonctionne, mais il n'est pas encore suffisamment robuste.

Nous allons maintenant ajouter une étape de validation :



4. Exercice 2 / Comprendre la validation Laravel

Laravel permet de définir des règles directement dans le contrôleur.

Par exemple :

```
$request->validate([
    'name' => 'required|string|max:100',
    'email' => 'required|email',
]);
```

Chaque champ possède une ou plusieurs règles.

Quelques règles importantes

Règle	Signification
`required`	Le champ est obligatoire
`string`	La valeur doit être une chaîne de caractères
`integer`	La valeur doit être un entier
`numeric`	La valeur doit être numérique
`email`	La valeur doit être une adresse e-mail valide
`max:100`	Maximum 100 caractères
`min:2`	Minimum 2 caractères
`nullable`	Le champ peut être vide

Les règles peuvent être combinées :

```
'name' => 'required|string|max:100'
```

Cela signifie :

> Le nom est obligatoire, doit être une chaîne de caractères et ne doit pas dépasser 100 caractères.

5. Exercice 3 / Ajouter la validation à la création d'un client

Ouvrez :

`app/Http/Controllers/ClientController.php`

Modifiez la méthode `store()`.

Utilisez les règles suivantes :

Champ	Règles
`name`	obligatoire, chaîne, maximum 100 caractères
`email`	obligatoire, e-mail valide
`phone`	facultatif, chaîne, maximum 20 caractères
`city`	facultatif, chaîne, maximum 100 caractères

Vous devez obtenir quelque chose de proche de :

```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');
}
```

Testez votre application

Essayez successivement :

Test 1

```
Nom : Jean Dupont
Email : jean@test.fr
```

Le client doit être créé.

Test 2

```
Nom :
Email : jean@test.fr
```

La création doit être refusée.

Test 3

```
Nom : Jean Dupont
Email : bonjour
```

La création doit être refusée.

Test 4

Saisissez un nom de plus de 100 caractères.

La création doit être refusée.

6. Que fait Laravel lorsqu'une validation échoue ?

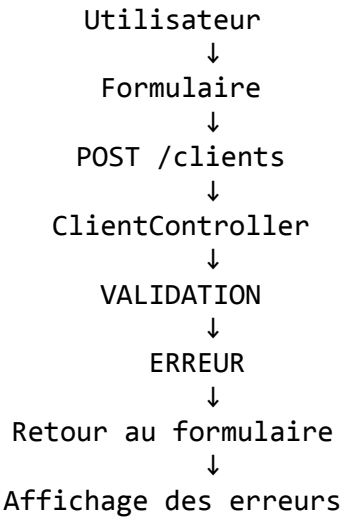
Lorsque cette instruction :

```php
$request->validate([
    'name' => 'required',
    'email' => 'required|email',
]);
```

détecte une erreur, Laravel :

1. arrête l'exécution du contrôleur ;
2. ne crée pas le client ;
3. retourne automatiquement vers le formulaire ;
4. transmet les erreurs à la vue ;
5. conserve les anciennes données saisies.

On obtient donc le fonctionnement suivant :



7. Exercice 4 / Afficher les erreurs

La validation est maintenant fonctionnelle.

Mais il y a un problème :

L'utilisateur sait-il réellement pourquoi son formulaire n'a pas été accepté ?

Nous allons afficher les erreurs dans Blade.

Ouvrez :

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

Ajoutez par exemple, au-dessus du formulaire :

```
@if ($errors->any())
    <div>
        <strong>Des erreurs sont présentes :</strong>

        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

Test

Essayez de créer un client sans nom et avec une adresse e-mail incorrecte.

Vous devez maintenant voir les erreurs retournées par Laravel.

8. Exercice 5 / Afficher une erreur sous chaque champ

L'affichage général fonctionne, mais une interface professionnelle doit indiquer directement **quel champ contient une erreur**.

Sous le champ `name`, ajoutez :

```
@error('name')
    <div>{{ $message }}</div>
@enderror
```

Sous le champ `email` :

```
@error('email')
    <div>{{ $message }}</div>
@enderror
```

Faites de même pour :

- `phone`
- `city`

Exemple

```
<label for="name">Nom</label>

<input
    type="text"
    name="name"
    id="name"
>

@error('name')
    <div>{{ $message }}</div>
@enderror
```

Question

Pourquoi est-il préférable d'afficher l'erreur directement sous le champ concerné ?

9. Exercice 6 / Conserver les données saisies

Imaginez que l'utilisateur remplisse un formulaire avec :

```
Nom : Jean Dupont
Email : jean.dupont
Ville : Paris
```

L'adresse e-mail est incorrecte.

Lorsque Laravel retourne sur le formulaire, il serait très pénible pour l'utilisateur de devoir tout saisir à nouveau.

Laravel permet de récupérer l'ancienne valeur avec :

```
{{ old('name') }}
```

Modifiez donc votre champ :

```
<input
    type="text"
    name="name"
    id="name"
    value="{{ old('name') }}"
>
```

Pour l’e-mail :

```
<input
    type="email"
    name="email"
    id="email"
    value="{{ old('email') }}"
>
```

Faites de même pour les autres champs.

Test

1. Saisissez un nom.
2. Saisissez un e-mail incorrect.
3. Envoyez le formulaire.
4. Vérifiez que le nom est toujours présent après le retour sur le formulaire.

10. Exercice 7 / Améliorer les messages

Laravel fournit automatiquement des messages d’erreur.

Cependant, nous pouvons personnaliser les messages.

Par exemple :

```
$request->validate(
    [
        'name' => 'required|string|max:100',
        'email' => 'required|email',
    ],
    [
        'name.required' => 'Le nom du client est obligatoire.',
        'name.max' => 'Le nom ne peut pas dépasser 100 caractères.',
        'email.required' => 'L’adresse e-mail est obligatoire.',
        'email.email' => 'L’adresse e-mail n’est pas valide.',
    ]
);
```

Test

Essayez de laisser le nom vide.

Vous devez maintenant obtenir :

```
Le nom du client est obligatoire.
```

11. Exercice 8 / Valider la modification d’un client

Nous avons validé la création. Mais un utilisateur peut également modifier un client.

Il faut donc appliquer les mêmes règles dans `update()`.

Modifiez :

```
public function update(Request $request, Client $client)
{
    // validation

    // modification

    // redirection
}
```

Ajoutez les règles :

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

Puis conservez le code permettant de modifier le client :

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

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

La méthode complète doit donc suivre cette logique :

```
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');
}
```

12. Exercice 9 / Vérification finale

Votre application doit maintenant respecter les règles suivantes.

Création d'un client

Situation	Résultat attendu
Nom vide	Erreur
Nom > 100 caractères	Erreur
E-mail vide	Erreur
E-mail invalide	Erreur
Téléphone vide	Autorisé
Ville vide	Autorisé
Toutes les données correctes	Création

Modification d'un client

Les mêmes règles doivent être appliquées.

13. Exercice final / Validation du produit

Nous allons maintenant appliquer ce que nous venons d'apprendre à une autre partie du Mini-CRM.

Dans le TP4, vous avez créé un CRUD pour les produits.

Un produit possède par exemple :

```
name
price
stock
```

Définissez les règles suivantes :

Champ	Règles
`name`	obligatoire, chaîne, maximum 150 caractères
`price`	obligatoire, numérique, supérieur ou égal à 0
`stock`	obligatoire, entier, supérieur ou égal à 0

Travail demandé

Ajoutez la validation :

- dans `store()` ;
- dans `update()`.

Puis affichez les erreurs dans le formulaire Blade.

Exemple de règles

```php
$request->validate([
    'name' => 'required|string|max:150',
    'price' => 'required|numeric|min:0',
    'stock' => 'required|integer|min:0',
]);
```

Tests à effectuer

Essayez notamment :

```
Nom :
Prix : 15
Stock : 10
```

Puis :

```
Nom : Clavier
Prix : -10
Stock : 10
```

Puis :

```
Nom : Clavier
Prix : 15
Stock : -5
```

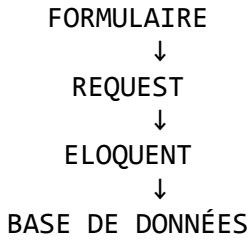
Puis :

```
Nom : Clavier
Prix : quinze
Stock : dix
```

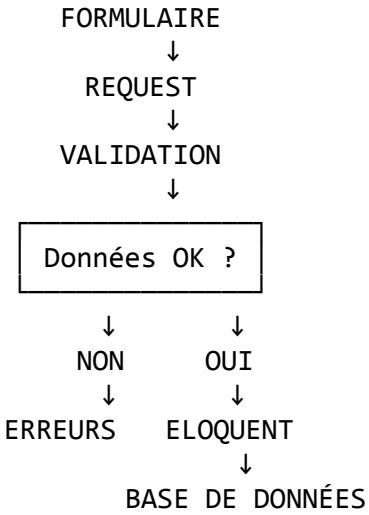
Enfin, testez un produit valide.

14. Bilan du TP

Au début du TP, nous avions :



Nous avons ajouté une étape essentielle :



Les principales notions à retenir

1. Valider les données

```
$request->validate([...]);
```

2. Définir des règles

```
'name' => 'required|string|max:100'
```

3. Afficher toutes les erreurs

```
@foreach ($errors->all() as $error)
    {{ $error }}
@endforeach
```

4. Afficher l'erreur d'un champ

```
@error('name')
    {{ $message }}
@enderror
```

5. Récupérer une ancienne valeur

```
{{ old('name') }}
```

À retenir

> **Un formulaire ne doit jamais envoyer directement ses données vers la base de données.**

Le serveur doit toujours :

recevoir → contrôler → valider → enregistrer

Au prochain TP, nous améliorerons encore cette architecture en sortant progressivement la logique de validation du contrôleur grâce aux **Form Requests**.

TP5 - Sujet complété

Annexe de réponses aux pages 1 à 13

Le sujet du professeur est conservé intégralement dans les 13 premières pages. Il ne contient aucun champ PDF remplissable. Cette annexe donne les réponses et les résultats associés à chaque exercice.

Pages 1-2 | Objectif et situation initiale

La validation agit **sur le serveur**, entre la Request et l'écriture Eloquent. Les attributs HTML required, type=email ou min sont utiles pour l'interface, mais une requête peut contourner le navigateur. Une donnée rejetée ne doit pas arriver à Client::create() ni à Product::create().

Exercice 1 | Vérification du problème

Le TP4 avait déjà une validation légère pour name, email et city. Le point de départ du PDF avec Client::create($request->all()) ne correspond donc pas exactement au projet existant. Le TP5 a renforcé les règles et ajouté phone facultatif ainsi que des messages visibles. À la fin du TP5, un nom vide, un email invalide et un nom trop long sont refusés. Le téléphone est une chaîne de 20 caractères maximum : les lettres y sont autorisées par la règle écrite du sujet.

Exercice 2 | Sens des règles

Règle	Signification
required	Une valeur doit être fournie.
string	La valeur doit être une chaîne de caractères.
integer	La valeur doit être un entier.
numeric	La valeur doit être numérique, y compris décimale.
email	La valeur doit être une adresse email valide.
max:100	Au plus 100 caractères pour une chaîne.
min:2	Au moins 2 caractères pour une chaîne.
nullable	La valeur peut être vide/null.

Exercice 3 | Validation lors de la création du client

Champ	Règles appliquées	Essais demandés
name	required, string, max:100	Nom vide ou 101 caractères : rejet.
email	required, email, unique	Email vide, incorrect ou déjà utilisé : rejet.
phone	nullable, string, max:20	Vide : accepté. Plus de 20 caractères : rejet.
city	nullable, string, max:100	Vide : accepté. Plus de 100 caractères : rejet.

Une saisie valide crée le client via **Client::create($this->validatedData($request))**. La méthode validatedData() appelle $request->validate(), puis le code utilise uniquement les valeurs validées. L'unicité de l'email est une protection supplémentaire cohérente avec la contrainte unique de la table clients.

Pages 5-8 | Erreurs, saisies et messages

Lors d'un échec, Laravel arrête la méthode, redirige vers le formulaire, transmet un sac d'erreurs à la session et remet les valeurs saisies à disposition via old(). Aucun client n'est créé.

Exercice 4 : une liste globale est affichée avec $errors->any(), $errors->all() et une boucle @foreach. Exercice 5 : chaque champ name, email, phone et city affiche son propre message avec @error('champ'). Exercice 6 : les champs utilisent old('champ') pour conserver la saisie. Exercice 7 : des messages français personnalisés sont fournis ; un nom vide affiche exactement « Le nom du client est obligatoire. »

Question page 7 | Pourquoi placer l'erreur sous le champ ?

L'utilisateur voit immédiatement **quel champ corriger** et comprend le problème sans devoir faire la correspondance avec une liste éloignée. Cela réduit les erreurs de correction et les nouveaux envois du formulaire.

Exercice 8 | Validation lors de la modification

update() réutilise les mêmes règles que store(). Le nom et l'email sont obligatoires ; phone et city sont facultatifs. Une mise à jour invalide n'écrit rien. La règle d'unicité ignore l'enregistrement modifié, ce qui permet de conserver son email actuel.

Exercice 9 | Matrice finale client

Situation	Résultat attendu et vérifié
Nom vide / plus de 100 caractères	Erreur ; aucune écriture.
Email vide / invalide	Erreur ; aucune écriture.
Téléphone ou ville vides	Accepté ; colonnes nullable.
Données correctes	Création ou modification, puis redirection vers la liste.
Modification invalide	Erreur ; ancien enregistrement préservé.

Pages 10-11 | Exercice final Produit

Champ	Règles	Essais
name	required|string|max:150	Vide et plus de 150 caractères : rejet.
price	required|numeric|min:0	-10 et « quinze » : rejet ; 15,50 : accepté.
stock	required|integer|min:0	-5 et « dix » : rejet ; 10 : accepté.

Ces règles s'appliquent à **store()** et **update()**. Les vues de création et d'édition affichent une liste globale d'erreurs, un message sous chaque champ et les anciennes valeurs. La colonne price est passée à DECIMAL(10,2) pour conserver les prix décimaux admis par numeric.

Pages 12-13 | Bilan

Le parcours est : Formulaire → Request → Validation. Si les données sont incorrectes, Laravel renvoie les erreurs et les anciennes valeurs au formulaire. Si elles sont valides, Eloquent les enregistre dans SQLite, puis le Controller redirige vers la liste.

Contrôles effectués

La migration TP5 est appliquée. La suite Laravel comprend 7 tests réussis, incluant les refus de données invalides, l'absence d'écriture après erreur, la conservation des saisies, les messages français, l'acceptation des champs facultatifs et un prix décimal.