

Cours Laravel 8 – les bases – la validation

 laravel.sillo.org/cours-laravel-8-les-bases-la-validation/

September 13, 2020



Nous avons vu dans le chapitre précédent un scénario mettant en œuvre un formulaire. Nous n'avons imposé aucune contrainte sur les valeurs transmises. Dans une application réelle, il est toujours nécessaire de vérifier que ces valeurs correspondent à ce qu'on attend. Par exemple un nom doit comporter uniquement des caractères alphabétiques et avoir une longueur maximale ou minimale, une adresse email doit correspondre à un certain format...

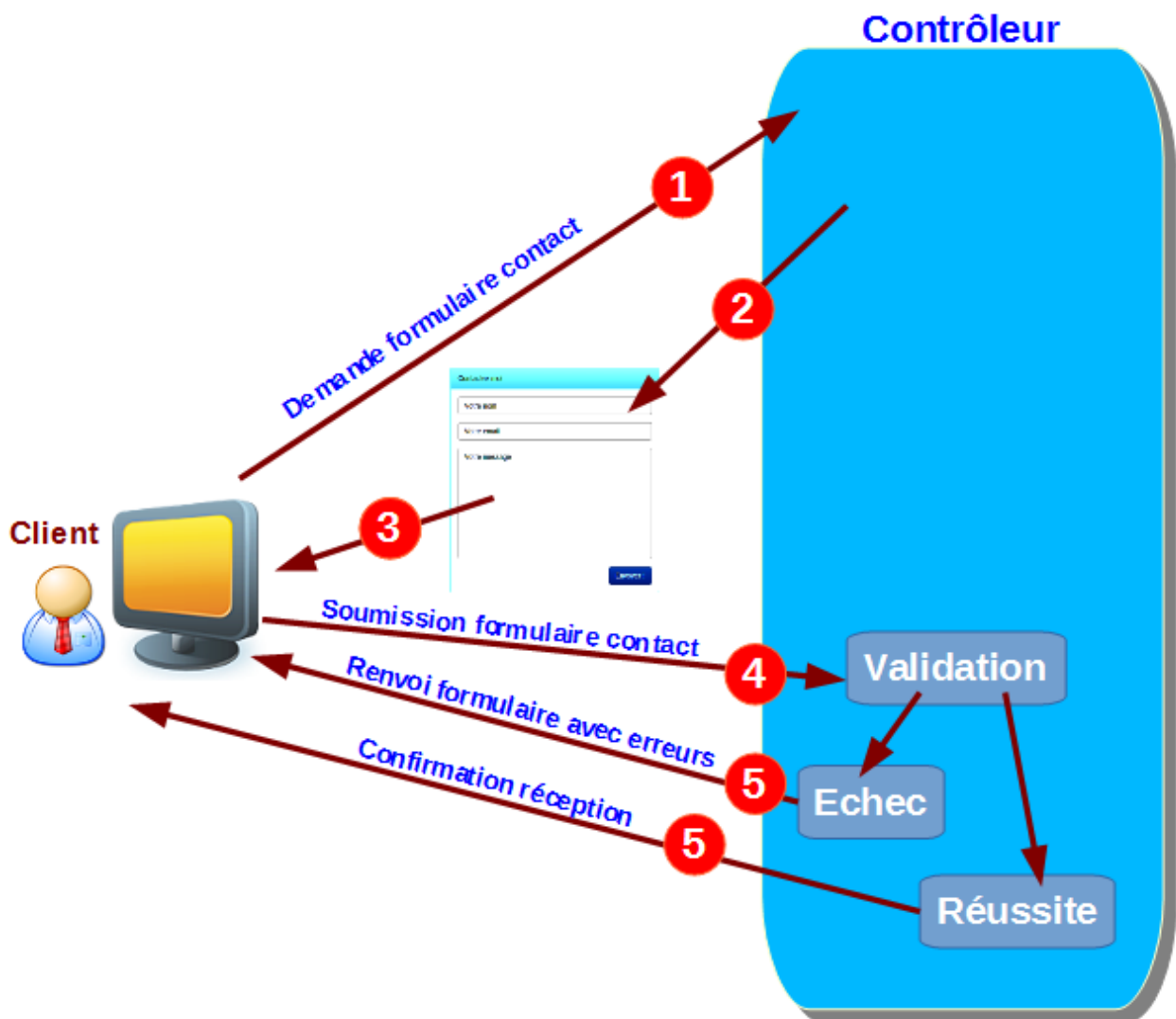
Il faut donc mettre en place des règles de validation. En général on procède à une première validation côté client pour éviter de faire des allers-retours avec le serveur. Mais quelle que soit la pertinence de cette validation côté client elle n'exonère pas d'une validation côté serveur.

On ne doit jamais faire confiance à des données qui arrivent sur le serveur !

Dans l'exemple de ce chapitre je ne prévoirai pas de validation côté client, d'une part ce n'est pas mon propos, d'autre part elle masquerait la validation côté serveur pour les tests.

Scénario et routes

Voici le scénario que je vous propose pour ce chapitre :



1. le client demande le formulaire de contact,
2. le contrôleur génère le formulaire,
3. le contrôleur envoie le formulaire,
4. le client remplit le formulaire et le soumet,
5. le contrôleur teste la validité des informations et là on a deux possibilités :
 - en cas d'échec on renvoie le formulaire au client en l'informant des erreurs et en conservant ses entrées correctes,
 - en cas de réussite on envoie un message de confirmation au client .

Routes

On va donc avoir besoin de 2 routes :

```
use App\Http\Controllers\ContactController;

Route::get('contact', [ContactController::class, 'create']);
Route::post('contact', [ContactController::class, 'store']);
```

On aura une seule url (avec verbe « get » pour demander le formulaire et verbe « post » pour le soumettre) :

Les vues

Le template

Pour ce chapitre je vais créer un template réaliste avec l'utilisation de Bootstrap pour alléger le code. Voici le code de ce template (**resources/views/template.blade.php**) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-
fit=no">
    <title>Mon joli site</title>
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
integrity="sha384-
JcKb8q3iqJ61gNV9KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z"
crossorigin="anonymous">
    <style>
      textarea { resize: none; }
      .card { width: 25em; }
    </style>
  </head>
  <body>
    @yield('contenu')
  </body>
</html>
```

J'ai prévu l'emplacement **@yield** nommé « contenu » pour recevoir les pages du site, pour notre exemple on aura seulement la page de contact et celle de la confirmation.

La vue de contact

La vue de contact va contenir essentiellement un formulaire (**resources/views/contact.blade.php**) :

```

@extends('template')

@section('contenu')
    <br>
    <div class="container">
        <div class="row card text-white bg-dark">
            <h4 class="card-header">Contactez-moi</h4>
            <div class="card-body">
                <form action="{{ { url('contact') } }}" method="POST">
                    @csrf
                    <div class="form-group">
                        <input type="text" class="form-control @error('nom') is-invalid @enderror"
name="nom" id="nom" placeholder="Votre nom" value="{{ old('nom') }}">
                        @error('nom')
                            <div class="invalid-feedback">{{ $message }}</div>
                        @enderror
                    </div>
                    <div class="form-group">
                        <input type="email" class="form-control @error('email') is-invalid @enderror"
name="email" id="email" placeholder="Votre email" value="{{ old('email') }}">
                        @error('email')
                            <div class="invalid-feedback">{{ $message }}</div>
                        @enderror
                    </div>
                    <div class="form-group">
                        <textarea class="form-control @error('message') is-invalid @enderror"
name="message" id="message" placeholder="Votre message">{{ old('message') }}
</textarea>
                        @error('message')
                            <div class="invalid-feedback">{{ $message }}</div>
                        @enderror
                    </div>
                    <button type="submit" class="btn btn-secondary">Envoyer !</button>
                </form>
            </div>
        </div>
    </div>
@endsection

```

Cette vue étend le template vu ci-dessus et renseigne la section « contenu ». Je ne commente pas la mise en forme spécifique à Bootstrap.

En cas de réception du formulaire suite à des erreurs on reçoit une variable **\$errors** qui contient un tableau avec comme clés les noms des contrôles et comme valeurs les textes identifiant les erreurs.

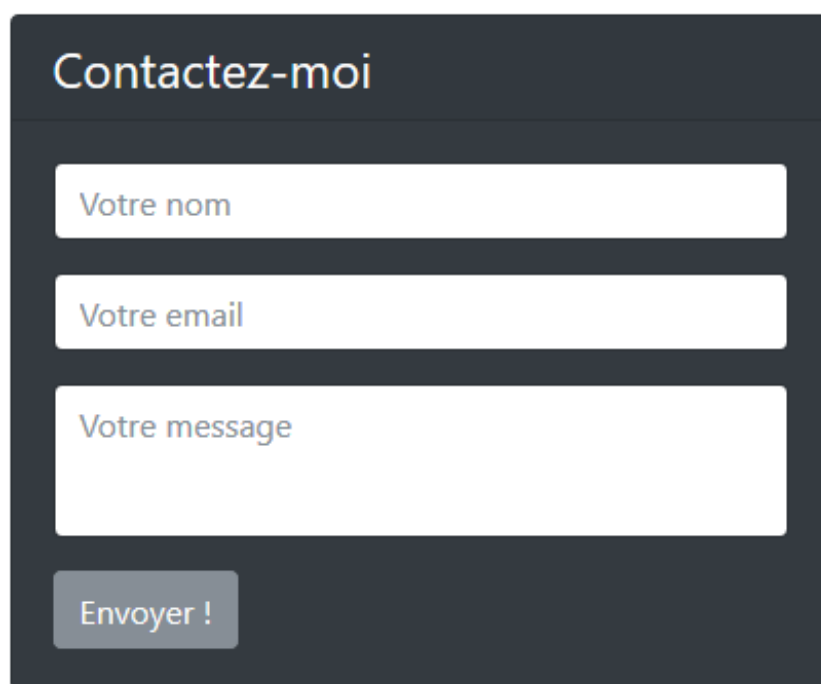
*La variable **\$errors** est générée systématiquement pour toutes les vues.*

On pourrait accéder à cette variable et faire un traitement spécifique pour afficher des erreurs mais Blade nous propose une possibilité plus élégante avec la directive **@error**.

Enfin en cas d'erreur de validation les anciennes valeurs saisies sont retournées au formulaire et récupérées avec l'helper **old** :

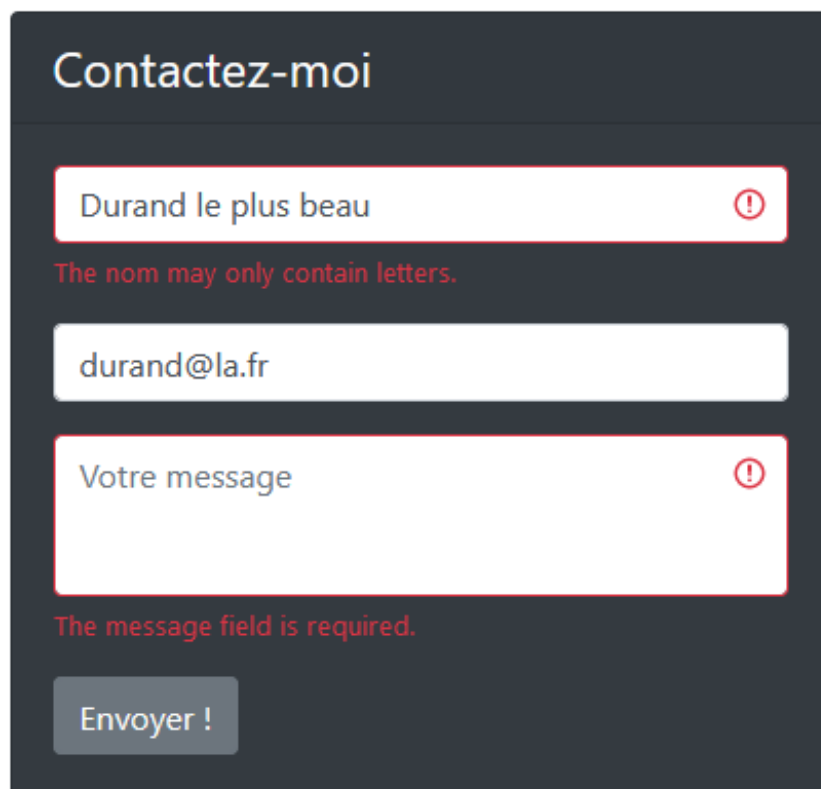
```
value="{{ old('nom') }}"
```

Au départ le formulaire se présentera ainsi :



A contact form titled "Contactez-moi" with a dark grey background. It contains three white input fields: "Votre nom", "Votre email", and "Votre message". Below the fields is a grey button labeled "Envoyer !".

Après une soumission et renvoi avec des erreurs il pourra se présenter ainsi :



The same contact form as above, but with validation errors. The "Votre nom" field contains "Durand le plus beau" and has a red border and a red exclamation mark icon. Below it is the error message "The nom may only contain letters." in red. The "Votre email" field contains "durand@la.fr". The "Votre message" field is empty and has a red border and a red exclamation mark icon. Below it is the error message "The message field is required." in red. The "Envoyer !" button remains at the bottom.

Les messages en français

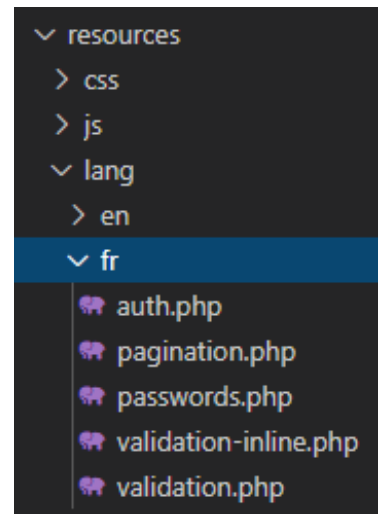
Par défaut les messages sont en anglais. Pour avoir ces textes en français vous devez

recupérer les fichiers [ici](#). Placez le dossier « fr » et son contenu dans le dossier **resources/lang** :

Ensuite changez cette ligne dans le fichier **config/app.php** :

```
'locale' => 'fr',
```

Vous devriez avoir vos messages en français :



Contactez-moi

!

Le champ nom doit contenir uniquement des lettres.

Votre message

!

Le champ message est obligatoire.

Envoyer !

La vue de confirmation

Pour la vue de confirmation (**resources/views/confirm.blade.php**) le code est plus simple et on utilise évidemment le même template :

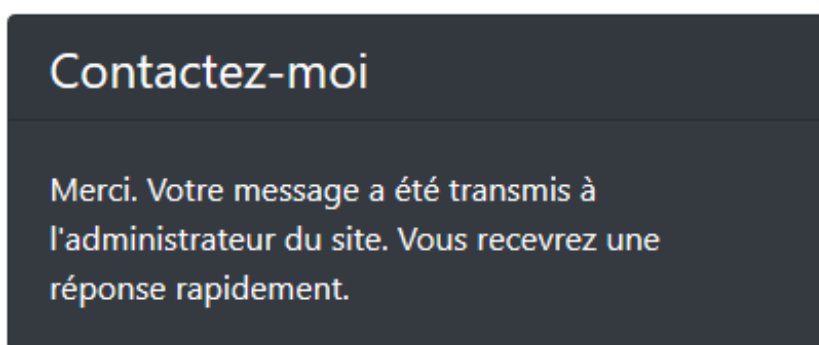
```

@extends('template')

@section('contenu')
    <br>
    <div class="container">
        <div class="row card text-white bg-dark">
            <h4 class="card-header">Contactez-moi</h4>
            <div class="card-body">
                <p class="card-text">Merci. Votre message a été transmis à l'administrateur du
site. Vous recevrez une réponse rapidement.</p>
            </div>
        </div>
    </div>
@endsection

```

Ce qui donne cette apparence :



La requête de formulaire

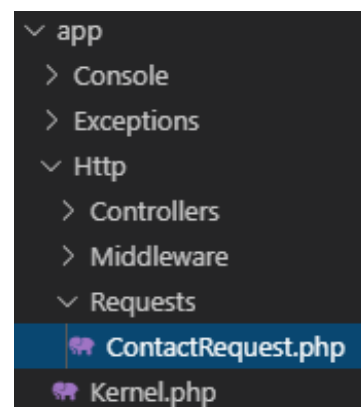
Il y a plusieurs façons d'effectuer la validation avec Laravel mais la plus simple et élégante consiste à utiliser une requête de formulaire (**Form request**).

Nous avons déjà utilisé Artisan qui permet d'effectuer de nombreuses opérations et nous allons encore avoir besoin de lui pour créer une requête de formulaire :

```
php artisan make:request ContactRequest
```

Comme par défaut le dossier n'existe pas il est créé en même temps que la classe :

Voyons le code généré :



```

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ContactRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}

```

La classe générée comporte 2 méthodes :

- **authorize** : pour effectuer un contrôle de sécurité éventuel sur l'identité ou les droits de l'émetteur,
- **rules** : pour les règles de validation.

On va arranger le code pour notre cas :


```

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ContactRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'nom' => 'bail|required|between:5,20|alpha',
            'email' => 'bail|required|email',
            'message' => 'bail|required|max:250'
        ];
    }
}

```

Au niveau de la méthode **rules** on retourne un tableau qui contient des clés qui correspondent aux champs du formulaire. Vous retrouvez le nom, l'email et le message. Les valeurs contiennent les règles de validation. Comme il y en a chaque fois plusieurs elles sont séparées par le signe « | ». Voyons les différentes règles prévues :

- **bail** : on arrête de vérifier dès qu'une règle n'est pas respectée,
- **required** : une valeur est requise, donc le champ ne doit pas être vide,
- **between** : nombre de caractères entre une valeur minimale et une valeur maximale,
- **alpha** : on n'accepte que les caractères alphabétiques,
- **email** : la valeur doit être une adresse email valide.

Au niveau de la méthode **authorize** je me suis contenté de renvoyer **true** parce que nous ne ferons pas de contrôle supplémentaire à ce niveau.

Vous pouvez trouver toutes les règles disponibles dans la documentation. Vous verrez que la liste est longue !

Le contrôleur

On va encore utiliser Artisan pour générer le contrôleur :

```
php artisan make:controller ContactController
```

Modifiez le code par défaut pour en arriver à celui-ci :

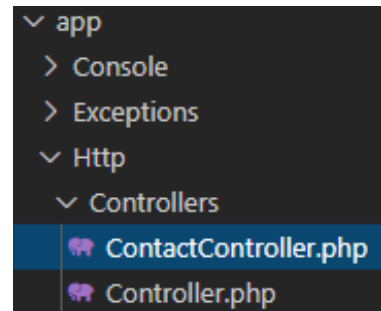
```
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ContactRequest;

class ContactController extends Controller
{
    public function create()
    {
        return view('contact');
    }

    public function store(ContactRequest $request)
    {
        return view('confirm');
    }
}
```



La méthode **create** ne présente aucune nouveauté par rapport à ce qu'on a vu au chapitre précédent. On se contente de renvoyer la vue **contact** qui comporte le formulaire.

La méthode **store** nécessite quelques commentaires. Vous remarquez le paramètre de type **ContactRequest**. On injecte dans la méthode une instance de la classe **ContactRequest** que l'on a précédemment créée. Laravel permet ce genre d'injection de dépendance au niveau d'une méthode. Je reviendrai en détail dans un prochain chapitre sur cette possibilité.

Si la validation échoue parce qu'une règle n'est pas respectée c'est la classe **ContactRequest** qui s'occupe de tout, elle renvoie le formulaire en complétant les contrôles qui étaient corrects et crée une variable **\$errors** pour transmettre les messages d'erreurs qu'on utilise dans la vue. Vous n'avez rien d'autre à faire !

Vérifiez avec la commande **php artisan route:list** que tout est correct :

```
λ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	contact		App\Http\Controllers\ContactController@create	web
	POST	contact		App\Http\Controllers\ContactController@store	web

On a bien nos deux routes avec l'url correct, le bon contrôleur avec les méthodes prévues et le middleware **web** appliqué aux deux routes.

Faites quelques essais avec des erreurs de saisie pour voir le fonctionnement.

D'autres façons d'effectuer la validation

Si vous n'appréciez pas les requêtes de formulaire et leur côté « magique » vous pouvez effectuer la validation directement dans le contrôleur avec la méthode **validate**. Voici le contrôleur modifié en conséquence :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ContactController extends Controller
{
    public function create()
    {
        return view('contact');
    }

    public function store(Request $request)
    {
        $this->validate($request, [
            'nom' => 'bail|required|between:5,20|alpha',
            'email' => 'bail|required|email',
            'message' => 'bail|required|max:250'
        ]);

        return view('confirm');
    }
}
```

Cette fois on injecte dans la méthode **store** directement la requête (**Illuminate\Http\Request**). Le fonctionnement est exactement le même.

Si cette méthode **validate** est encore trop abstraite à votre goût vous pouvez détailler les opérations :

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class ContactController extends Controller
{
    public function create()
    {
        return view('contact');
    }

    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'nom' => 'bail|required|between:5,20|alpha',
            'email' => 'bail|required|email',
            'message' => 'bail|required|max:250'
        ]);

        if ($validator->fails()) {
            return back()->withErrors($validator)->withInput();
        }

        return view('confirm');
    }
}

```

On utilise la façade **Validator** en précisant toutes les entrée (**\$request->all()**) et les règles de validation. Ensuite si la validation échoue (**fails**) on renvoie (**back**) le formulaire avec les erreurs (**withErrors**) et les valeurs entrées (**withInput**) pour pouvoir les afficher dans le formulaire.

Mais pourquoi se compliquer la vie quand on dispose de fonctionnalités plus simples et élégantes ?

En résumé

- La validation est une étape essentielle de vérification des entrées du client.
- On dispose de nombreuses règles de validation.
- Le validateur génère des erreurs explicites à afficher au client.
- Pour avoir les textes des erreurs en français il faut aller chercher les traductions et les placer dans le bon dossier.
- Les requêtes de formulaires (**Form request**) permettent d'effectuer la validation de façon simple et élégante.
- Il y a plusieurs façons d'effectuer la validation à adapter selon les goûts et les circonstances.