Validation des données d’entrée:

1.Introduction :

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.

2.Validation manuelle

```
<?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)
{
```

```
$rules=[
'name' => ['required','between:5,20'],
'mail' => ['required','email'],
'section' => ['required','max:250'],
'picture'=>['required','max:1024','image'],

] ;
 $messages=[
            'name.required'=>'please write your name',
            'picture.max'=>'this picture is greater than 1024 kb'
         ];

$validator = Validator::make($request->all(),$rules ,$messages
);
if ($validator->fails()) {
return back()->withErrors($validator)->withInput();
}
 //save data
         $path_image=$request->picture->store('students','public');
         $student= new Student();
         $student->name=$request->name;
         $student->mail=$request->mail;
         $student->section=$request->section;
         $student->picture=$path_image;
         $student->save();
         session()->flash('success', 'student saved successfully');
         return redirect()->route('students.create');
}
}
```

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.

3.Arrêt au premier échec de validation

Il peut arriver que vous souhaitiez arrêter l'exécution des règles de validation sur un attribut après le premier échec de validation. Pour ce faire, affectez la règle de validation à l'attribut :

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
```

```
]);
```

Dans cet exemple, si la règle ***unique*** sur l'attribut ***title*** échoue, la règle ***max*** ne sera pas vérifiée. Les règles seront validées dans l'ordre où elles sont attribuées.

4.Affichage des erreurs de validation

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.*

Laravel met aussi à notre disposition la directive Blade ***@error("nom_champ")*** pour vérifier s'il existe une erreur de validation pour le champ ***nom_champ*** :

```
@error('name')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Dans la directive ***@error***, nous avons accès au message d'erreur à travers la variable ***$message***.

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

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

5.Personnalisation des messages d'erreur

Si nécessaire, vous pouvez fournir des messages d'erreur personnalisés qu'une instance de validateur doit utiliser à la place des messages d'erreur par défaut fournis par Laravel. Il existe plusieurs façons de spécifier des messages personnalisés. Premièrement, vous pouvez passer les messages personnalisés comme troisième argument à la méthode ***Validator::make*** :

```
$validator = Validator::make($input, $rules, $messages = [
    'email.required' => 'We need to know your email address!',
```

```
]);
```

6. Afficher une erreur Laravel au formulaire Bootstrap

Pour afficher les erreurs d'un formulaire avec Bootstrap, nous avons :

1. La classe CSS **.is-invalid** qu'il faut ajouter à la classe CSS .form-control d'un champ <input>, <textarea>, ... pour y indiquer la présence d'erreur
2. La classe CSS .**invalid-feedback** qu'il faut ajouter sur un autre élément HTML (, , ...) pour afficher le message d'erreur

Ainsi, nous pouvons vérifier la présence d'erreurs sur un input en utilisant la directive ***@error*** ou les méthodes de l'objet ***$errors*** pour ajouter/retirer la classe CSS **.is-invalid** qui va déclencher l'affichage/masquage de l'élément de classe CSS **.invalid-feedback**.

Le code initial de la vue **resources/views/contact.blade.php** devient :

```
<input type="email" class="form-control @error('email') is-invalid
@enderror" id="email" name="email" >

            <!-- Le message d'erreur -->
            @error('email')
            <div class="invalid-feedback">{{ $message }}</div>
            @enderror
```

7. Liste des règles de validation disponibles

Voici une liste de toutes les règles de validation disponibles dans Laravel que vous pouvez utiliser.

Règle	Exemple	Description
accepted	'field' => 'accepted'	Useful for checkbox validation it must be yes, on, 1 or true

active_url	'field' => 'active_url'	The field under validation must have a valid A or AAAA record according to the dns_get_record PHP function
after:date	'field' => 'date|after:tomorrow'	The field under validation must be a value after a given date
after_or_equal:date	'field' => 'date|after_or_equal:other_field'	The field under validation must be a value after or equal to the given date.
alpha	'field => 'required|alpha'	The field under validation must be entirely alphabetic characters.
alpha_dash	'field' => 'required|alpha_dash'	The field under validation may have alpha-numeric characters, as well as dashes and underscores.
alpha_num	'field' => 'required|alpha_num'	The field under validation must be entirely alpha-numeric characters.
array	'field' => 'array'	The field under validation must be a PHP array.
bail	'field' => 'bail|required'	Stop running validation rules after the first validation failure.
before:date	'field' => 'date|before:tomorrow'	The field under validation must be a value preceding the given date.
before_or_equal:date	'field' => 'date|before_or_equal:other_field'	The field under validation must be a value preceding or equal to the given date.
between:min,max	'field' => 'required|between:1,10'	The field under validation must have a size between the given *min* and *max*.
boolean	'field' => 'boolean'	The field under validation must be able to be cast as a boolean. Allowed: true, false, 1, 0, "1", and "0".
confirmed	'field' => 'confirmed'	If the field under validation is password, a matching password_confirmation field must be present.
date	'field' => 'date'	The field under validation must be a valid, non-relative date according to the strtotime PHP function.
date_equals:date	'field' => 'date|date_equals:compare_field'	The field under validation must be equal to the given date.
date_format:format	'field' => 'date|date_format:YYYY:mm:dd'	The field under validation must match the given *format*.
different:field	'field' => 'different:compare_field'	The field under validation must have a different value than *field*.
digits:value	'field' => 'digits:10'	The field under validation must be *numeric* and must have an exact length of *value*.
digits_between:min,max	'field' => 'digits_between:0,10'	The field under validation must have a length between the given *min* and *max*.
dimensions	'avatar' => 'dimensions:min_width=100,min_height=200'	The file under validation must be an image meeting the dimension constraints as specified by the rule's parameters: Available constraints are: *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*.
distinct	'foo.*.id' => 'distinct'	When working with arrays, the field under validation must not have any duplicate values.
email	'field' => 'email'	The field under validation must be formatted as an e-mail address.
ends_with:word1,word2	'field' => 'ends_with:foo,bar'	The field under validation must end with one of the given values.
exists:table,column	'country' => 'exists:countries'	The field under validation must exist on a given database table.

file	'field' => 'file'	The field under validation must be a successfully uploaded file.
filled	'field' => 'filled'	The field under validation must not be empty when it is present.
gt:field	'field' => 'gt:100'	The field under validation must be greater than the given *field*.
gte:field	'field' => 'gte:100'	The field under validation must be greater than or equal to the given *field*.
image	'field' => 'image'	The file under validation must be an image (jpeg, png, bmp, gif, svg, or webp)
in:foo,bar	'field' => 'in:foo,bar'	The field under validation must be included in the given list of values.
integer	'field' => 'integer'	The field under validation must be an integer.
ip	'field' => 'ip'	The field under validation must be an IP address.
json	'field' => 'json'	The field under validation must be a valid JSON string.
lt:field	'field' => 'lt:100'	The field under validation must be less than the given *field*.
lte:field	'field' => 'lte:100'	The field under validation must be less than or equal to the given *field*.
max:value	'field' => 'max:100'	The field under validation must be less than or equal to a maximum *value*.
mimetypes:text/plain	'photo' => 'mimes:jpeg,bmp,png'	The file under validation must match one of the given MIME types
min:value	'field' => 'min:100'	The field under validation must have a minimum *value*.
not_in:foo,bar	'field' => 'not_in:foo,bar'	The field under validation must not be included in the given list of values.
not_regex:pattern	'email' => 'not_regex:/^.+$/i'	The field under validation must not match the given regular expression.
nullable	'field' => 'nullable'	The field under validation may be null.
numeric	'field' => 'numeric'	The field under validation must be numeric.
present	'field' => 'present'	The field under validation must be present in the input data but can be empty.
regex:pattern	'email' => 'regex:/^.+$/i'	The field under validation must match the given regular expression.
required	'field' => 'required'	The field under validation must be present in the input data and not empty.
string	'field' => 'string'	The field under validation must be a string.
unique:table,column,except,id	'email' => 'unique:users,email_address'	The field under validation must not exist within the given database table.
url	'field' => 'url'	The field under validation must be a valid URL.
uuid	'field' => 'uuid'	The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) UUID.