

La validation en Symfony

Table des matières

I. Contexte	3
II. La validation à partir de l'entité	3
A. La validation à partir de l'entité.....	3
B. Exercice : Quiz.....	9
III. Autres validations de formulaires et contrôleur	10
A. Validation dans les classes de formulaires.....	10
B. Validation dans le contrôleur	11
C. Validation dans un fichier Twig.....	15
D. Exercice : Quiz.....	15
IV. Essentiel	16
V. Auto-évaluation	17
A. Exercice	17
B. Quiz	17
Solutions des exercices	19

I. Contexte

Durée : 1 h

Environnement de travail : PC

Prérequis :

- Notion de PHP
- Notion de POO
- Connaître les formulaires avec Symfony

Contexte

Afin de dialoguer avec un utilisateur, il est nécessaire d'utiliser des formulaires. Ceux-ci permettent de pouvoir s'authentifier, répondre à des questionnaires et bien d'autres choses. Avec Symfony, il existe deux façons de créer un formulaire. Soit à partir d'une classe `FormType` générée en amont (notamment via le `MakerBundle`), ou directement via le `formBuilder` accessible dans un contrôleur ; soit à partir d'une classe `FormType` générée en amont (via le `MakerBundle`) ou directement via le `formBuilder` accessible dans un contrôleur.

Après quoi, il faudra que le contrôleur retourne le formulaire à la Vue (Modèle-Vue-Contrôleur), par exemple dans un fichier Twig afin qu'il puisse être visualisé, rempli, soumis par l'utilisateur puis validé par le contrôleur. Ainsi, avant même de comprendre ce qu'est la validation avec Symfony, qui est le thème de ce cours, il vous faudra avoir acquis ces quelques notions sur les formulaires.

Pourquoi la validation est-elle une étape importante ? Il existe toutes sortes d'utilisateurs, que ce soit des personnes inexpérimentées qui peuvent saisir des données, ou d'autres plus expérimentées qui peuvent être malveillantes et compromettre la sécurité de l'application ou de la base de données.

La validation consiste à vérifier que les réponses données soient conformes à celles attendues et puissent être affectées à vos entités puis stockées en BDD (Base De Données) avec toutes les garanties. Alors, quels outils doit-on utiliser pour la validation avec Symfony ? Existe-t-il plusieurs façons de procéder ? Concrètement, comment s'y prendre ? C'est ce que nous analyserons ensemble dans ce cours.

II. La validation à partir de l'entité

A. La validation à partir de l'entité

Les entités sont des éléments importants de Symfony, car l'ORM (*Object-Relational Mapping*) du framework se base sur celles-ci pour construire la base de données. Une entité peut par exemple représenter une table, ses propriétés, ses champs ou encore ses attributs. Ainsi, chaque saisie représente généralement un objet qui sera stocké dans la base de données. C'est pourquoi il est approprié de placer les contraintes de validation dans les entités.

Méthode

Installation de bundles

Afin de nous aider, Symfony nous fournit un composant qu'il est nécessaire d'installer à l'aide du terminal de commandes :

```
1 $ composer require symfony/validator doctrine/annotations
```

La commande « **composer require symfony/validator doctrine/annotations** » permet d'installer les packages « **symfony/validator** » et « **doctrine/annotations** » en utilisant « **Composer** », un gestionnaire de dépendances pour PHP.

Cela signifie que vous pouvez utiliser les fonctionnalités et les classes de ces packages dans votre projet. « **symfony/validator** » fournit un système de validation de données pour les applications PHP et « **doctrine/annotations** » permet de définir les annotations dans le code pour l'utilisation avec l'ORM Doctrine.

En utilisant « **doctrine/annotations** », vous pouvez ajouter des informations à votre code sans avoir à écrire de code supplémentaire ou à configurer des fichiers externes. Ce qui est important de comprendre c'est que, sous le terme « **annotations** », Symfony inclut le système d'annotations, mais également **les attributs** qui ont été introduits depuis PHP 8, car ces métadonnées sont natives à PHP.

Ainsi, nous n'utilisons que des attributs de « **PHP attributes** » dans ce cours.

Méthode Importer un namespace

Ensuite, on importe un namespace en utilisant la commande suivante :

```
1 $ use Symfony\Component\Validator\Constraints as Assert;
```

Grâce à cet import, on peut accéder aux classes utilisant l'alias Assert. Par exemple, au lieu de devoir écrire « **new Symfony\Component\Validator\Constraints\NotBlank()** », on écrira « **new Assert\NotBlank()** ». La classe « **Constraints** », qui utilise l'alias Assert, nous permettra ainsi d'ajouter des contraintes de validation.

Exemple

```
1 <?php
2
3 namespace App\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6 use Symfony\Component\Validator\Constraints as Assert;
7 use App\Repository\TaskRepository;
8 use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
9
10 #[ORM\Entity(repositoryClass: TaskRepository::class)]
11 class Task
12 {
13
14
15     #[Assert\NotBlank]
16     private $name;
17 }
```

La déclaration « **#[Assert\NotBlank]** » permet d'ajouter une contrainte de validation sur une propriété d'une entité dans un projet Symfony.

Remarque

Il est possible, au lieu de rajouter des contraintes directement sur l'entité, de les mettre à part dans un fichier YAML ou XML dans le dossier config.

Exemple

```
1 # config/validator/validation.yaml
2 App\Entity\Task:
3   properties:
4     name:
5       - NotBlank: ~
```

Codes des vidéos disponibles en téléchargement ici :

[cf. Projet8-Todolist-main.zip]

Les contraintes de validation

Les contraintes de validation sont utilisées pour vérifier la validité des données entrées dans une entité dans le cadre d'un formulaire. Elles permettent de contrôler l'entité elle-même ainsi que ses variables, qui sont représentées par les champs de saisie. Symfony offre de nombreuses contraintes de base, mais il est également possible d'importer des contraintes supplémentaires à partir de paquets tiers ou de créer ses propres contraintes personnalisées.

Voici quelques-unes des contraintes les plus utilisées avec leurs significations suivies d'un exemple d'utilisation avant une propriété :

Contraintes	Description	Exemple
NotBlank	Vérifie que la valeur soumise n'est ni une chaîne de caractères vide ni nulle.	<code>#[Assert\NotBlank]</code>
Blank	Inverse de NotBlank.	<code>#[Assert\Blank]</code>
NotNull	Vérifie qu'une valeur n'est pas strictement nulle (à la différence de NotBlank, peut être vide).	<code>#[Assert\NotNull]</code>
IsNull	Inverse de NotNull.	<code>#[Assert\Null]</code>
IsTrue	Vérifie que la valeur est True, possède la valeur integer 1 ou string '1'.	<code>#[Assert\IsTrue]</code>
IsFalse	Vérifie que la valeur est False, possède la valeur integer 0 ou string '0'.	<code>#[Assert\Isfalse]</code>
Type	Vérifie que le type de la donnée correspond au type voulu.	<code>#[Assert\Type(Address::Integer)]</code>
Email	Vérifie que c'est une adresse email valide.	<code>#[Assert\Email]</code>
length	Vérifie que la longueur d'un string se situe entre min et max. Les options minMessage et maxMessage peuvent utiliser le paramètre limit qui correspond au min ou max.	<code>#[Assert\Length(min: 2, max: 50, minMessage: 'Vous devez avoir plus de {{ limit}} caractère')]</code>
Url	Vérifie que la valeur entrée est une adresse URL valide.	<code>#[Assert\URL]</code>

Contraintes	Description	Exemple
Regex	Valide qu'une valeur correspond à une expression régulière.	<code>#[Assert\Regex("/^[0-9]{5}\$/")]</code>
UserPassword	Valide que le mot de passe est valide. Notez qu'il faut importer une autre classe « <i>use Symfony\Component\Security\Core\Validator\Constraints as SecurityAssert;</i> ».	<code># [SecurityAssert\UserPassword]</code>
Contraintes de comparaison	Valide si la valeur est égale à l'option. <code>NotEqualTo()</code> étant l'inverse.	<code>#[Assert\EqualTo('John')]</code>
	Comme <code>EqualTo</code> , mais dans le sens strict. Équivaut à <code>===</code> et <code>EqualTo ==</code> .	<code>#[Assert\IdenticalTo(3)] # [Assert\NotIdenticalTo(3)]</code>
	Valide qu'une valeur est strictement inférieure à la valeur définie dans l'option et <code>LessThanOrEqual()</code> ajoute l'égalité.	<code>#[Assert\LessThan(3)]</code>
	Valide qu'une valeur est strictement supérieure à la valeur définie dans l'option et <code>#[Assert\GreaterThanOrEqual(3)]</code> ajoute l'égalité.	<code>#[Assert\GreaterThan(3)]</code>
	Vérifie qu'un nombre ou un <code>DateTime</code> soit compris entre la valeur minimum et la valeur maximum donnée.	<code>#[Assert\Range(min: 100, max: 200)]</code>
	Vérifie que cette valeur n'est pas déjà utilisée pour cette propriété.	<code>#[Assert\Unique]</code>
Contraintes de nombre	Vérifie la valeur d'un nombre s'il est positif ou négatif, avec ou sans le zéro.	<ul style="list-style-type: none"> • <code>#[Assert\Positif]</code> • <code>#[Assert\PositifOrZero]</code> • <code>#[Assert\Négatif]</code> • <code>#[Assert\NégatifOrZero]</code>
Contraintes de Date	Valide que la valeur est ou peut être convertie en YYYY-MM-DD.	<code>#[Assert\Date]</code>

Contraintes	Description	Exemple
	Valide que la valeur est ou peut être convertie en DateTime.	#[Assert\DateTime]
	Valide que la valeur est ou peut être convertie en HH:MM:SS.	#[Assert\time]
	Valide que la valeur est identifiée en fuseau horaire (par exemple, Europe/paris).	#[Assert\TimeZone]
Choice	Valide que la valeur donnée existe dans un tableau référent.	#[Assert\Choice(['Actus', 'Divers', 'Retro'])]

Exemple

Dans la classe fictive ci-dessous, voici d'exemple d'intégration de ces contraintes de validation.

```

1 use Symfony\Component\Validator\Constraints as Assert;
2
3 class ExampleEntity
4 {
5     #[Assert\NotBlank]
6     #[Assert\Length(min: 2, max: 50, minMessage: 'Vous devez avoir plus de {{ limit }} caractères')]
7     #[Assert\EqualTo('John')]
8     private string $name;
9
10
11     #[Assert\NotNull]
12     #[Assert\Type('integer')]
13     #[Assert\Positive]
14     #[Assert\Regex("/^\d{2}$/")]
15     private int $age;
16
17     #[Assert\Null]
18     #[Assert\Negative]

```

```

19 private ?int $negativeNumber;
20
21 #[Assert\Email]
22 private string $email;
23
24 #[Assert\IsTrue]
25 private bool $trueValue;
26
27 #[Assert\IsFalse]
28 private bool $falseValue;
29
30 #[Assert\URL]
31 private string $url;
32
33 #[Assert\DateTime]
34 private \DateTimeInterface $dateTime;
35
36 #[Assert\Time]
37 private \DateTimeInterface $time;
38
39 #[Assert\Date]
40 private \DateTimeInterface $date;
41
42 #[Assert\TimeZone]
43 private string $timeZone;
44
45 #[Assert\IdenticalTo(3)]
46 #[Assert\NotIdenticalTo(3)]
47 #[Assert\LessThan(3)]
48 #[Assert\GreaterThan(3)]
49 #[Assert\Range(min: 100, max: 200)]
50 private int $number;
51
52 #[Assert\Unique]
53 #[Assert\Choice(['Actus', 'Divers', 'Retro'])]
54 private string $category;
55
56 #[Assert\Blank]
57 #[Assert\NegativOrZero]
58 #[Assert\PositiveOrZero]
59 private ?int $nullableNumber;
60
61 #[SecurityAssert\UserPassword]
62 private string $password;
63 }

```

Autre exemple, le tableau est défini dans la constante const et est utilisé par choices :

```

1 class Task
2 {
3     const CATEGORY = ['Actus', 'Divers', 'Retro']
4
5     #[Assert\Choice(choices: CATEGORY)]
6     protected $category;
7 }

```


Remarque

Notez que, pour toutes les contraintes, il est possible d'ajouter l'option message. Exemple :

```
# [Assert\Email (message: 'E-mail invalide') ]
```

B. Exercice : Quiz

[solution n°1 p.21]

Question 1

Il est possible de mettre des contraintes de validation sur un formulaire à partir d'un fichier XML.

- ☐ Vrai
- ☐ Faux

Question 2

Quelle proposition est fausse sur une propriété d'entité ?

- ☐ #[Assert\Email(message: "Veuillez écrire l'email au bon format")]
- ☐ #[Assert\NotBlank(message: "le champ être rempli")]
- ☐ Aucune des propositions

Question 3

La contrainte de validation isTrue est fausse si la réponse est une chaîne de caractère string, par exemple : '1'.

- ☐ Vrai
- ☐ Faux

Question 4

On peut mettre plus de 5 contraintes de validation sur une propriété.

- ☐ Vrai
- ☐ Faux

Question 5

Quelle réponse est exacte ?

- ☐ /***@Assert\Blank
- ☐ #[Assert\NotBlank]
- ☐ Les réponses 1 et 2 sont exactes

III. Autres validations de formulaires et contrôleur

A. Validation dans les classes de formulaires

Lorsque l'on crée une entité à l'aide de MakerBundle (c'est-à-dire une dépendance qui aide à installer divers éléments sur des projets Symfony), celui-ci crée également un formType qui sera situé dans le dossier Form. Ce formType contiendra un formulaire avec les champs nécessaires. Toutefois, il est également possible de créer un FormType « à la main ». Vous pourrez alors mettre vos contraintes de validation dans ce fichier.

Ainsi, dans la méthode buildForm(), les champs peuvent être définis par la méthode add() de FormBuilderInterface. C'est d'ailleurs dans cette méthode qu'il faudra ajouter les contraintes de validation plutôt que de les mettre dans l'entité.

Pour les mettre en place, il suffit de se rendre dans le troisième paramètre de la méthode add() qui est un tableau associatif d'options où l'on peut ajouter différentes contraintes de validation.

Prenons l'exemple d'un formulaire avec le code ci-dessous :

```

1 <?php
2 namespace App\Form;
3
4 use Symfony\Component\Form\AbstractType;
5 use Symfony\Component\Form\FormBuilderInterface;
6 use Symfony\Component\Validator\Constraints\Length;
7 use Symfony\Component\Validator\Constraints\NotBlank;
8 use Symfony\Component\Form\Extension\Core\Type\TextareaType;
9
10 class TaskType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array $options)
13     {
14         $builder
15             ->add('title', TextType::class, [
16                 'required' => true,
17                 'constraints' => [
18                     new NotBlank([
19                         'message' => 'Un titre doit être saisi',
20                     ]),
21                     new Length([
22                         'min' => 5,
23                         'minMessage' => 'Le titre doit contenir au moins {{ limit }}
24 caractères',
25                         'max' => 255,
26                     ]),
27                 ],
28             ->add('content', TextareaType::class)
29     }
30 }
```

Dans ce Code, nous constatons la présence de deux champs qui sont définis par les méthodes add() : « **'title'** » et « **'content'** ». Vous remarquerez d'ailleurs qu'un tableau a été ouvert dans le champ title. Il s'agit en effet d'un tableau d'options facultatif qui permet de spécifier des options supplémentaires pour le champ title, telles que la longueur maximale, la validation des données, etc., et qui peut contenir des contraintes de validation.

Dans notre exemple, le tableau d'option ouvert dans le champ title contient des contraintes de validation que l'on retrouve dans « **constraints** ». Celui-ci représente un tableau d'options supplémentaires pour le champ title. Il sert à définir des contraintes de validation pour ce champ, telles que « **NotBlank** », pour s'assurer qu'il n'est pas vide, et « **Length** », pour vérifier la longueur minimale et maximale du contenu.

De plus, chaque contrainte commence par le mot « **new** ». Le mot « **new** » est un opérateur de création d'objet en PHP. Il est utilisé pour instancier des objets à partir de classes. Si vous regardez bien en haut du code, les classes `NotBlank` et `Length` ont été importées. Ce sont des classes fournies par la bibliothèque de validation de formulaire `Symfony Validator`.

B. Validation dans le contrôleur

Maintenant, comment le contrôleur traite-t-il la validation du formulaire après soumission de celui-ci par l'utilisateur ?

En général, si un formulaire est associé à une entité (`data_type`) nous plaçons les contraintes directement dans l'entité.

Si le formulaire n'est pas associé à une entité (par exemple : formulaire de contact, DTO, etc.), on aura tendance à affecter celles-ci directement dans le builder du `formType`.

SensioLabs (le développeur de `Symfony`) préconise pour la version 5.4 de son framework d'utiliser la condition ci-dessous :

```
1 if ($form->isSubmitted() && $form->isValid()) {  
2     $userRepository->save($user, true);  
3  
4     return $this->redirectToRoute('app_user_index', [], Response::HTTP_SEE_OTHER);  
5 }
```

Bien sûr, « **\$form** » représente le formulaire qui a été défini, et si « **isValid()** », qui vérifie entre autres les contraintes, retourne « **true** » tout comme `isSubmitted()`, alors on entrera dans le traitement de la condition.

Notez que pour déboguer les contraintes, il existe cette ligne de commande qui peut vous être très utile :

```
1 $ php bin/console debug:validator 'App\Entity\nom_entité'
```

Autres contraintes de validation

Toutes les contraintes de validation ci-dessous sont mises en exemple avec des attributs par simplification et parce que cela prend moins de place.

Toutefois, comme nous l'avons vu dans cette partie du cours, dans un fichier PHP, que ce soit dans un `formType` ou dans un contrôleur, les contraintes sont mises dans un tableau « **constraints** » et chacune est déclarée par le mot « **new** » suivi du nom de la contrainte.

Vous pouvez inclure les classes de validation de formulaire nécessaires soit en les insérant manuellement, soit en utilisant l'autoload. Si vous voulez utiliser le mot clé « **Assert** », vous pouvez inclure la classe « **Constraints as Assert** » en autoload. Cela vous permettra d'écrire les contraintes de validation de la manière suivante : « **new Assert\File** ».

Par exemple, pour la contrainte de type « **File** », vous pouvez écrire soit « **new File** », soit « **new Assert\File** ». Les options pour cette contrainte peuvent alors être définies dans un tableau appelé « **constraints** ». Cela est valable bien sûr pour toutes les contraintes de validation décrites dans la partie « *La validation à partir de l'entité* ».

Remarque

Le site officiel de `Symfony` peut également vous apporter une aide précieuse : <https://symfony.com/doc/current/reference/constraints.html>

Contraintes de fichier :

File : valide que le fichier correspond aux options demandées.

```
1 #[Assert\File(
2     maxSize: '1024k',
3     mimeTypes: ['application/pdf', 'application/x-pdf'],
4     mimeTypeMessage: 'vérifiez que le format du fichier soit bien en PDF'
5 )]
```

Image : valide que le fichier correspond aux options demandées.

```
1 #[Assert\Image( //les options peuvent être nombreuses que ce soit sur le format ou la taille)]
```

Contraintes de finance

Il existe de nombreuses contraintes issues du monde de la finance. Pour qu'une contrainte soit respectée, il suffit d'ajouter l'attribut.

```
1 #[Assert\Bic] contraintes proposées par Symfony : Bic, CardScheme, Currency, Luhn, Iban, Isbn,
    Issn, Isin.
```

AtLeastOneOf : vérifie que la valeur répond à au moins une des contraintes déclarées.

```
1 #[Assert\AtLeastOneOf([
2     new Assert\Regex('/#/'),
3     new Assert\Length(min: 10),
4 ])]
```

Sequentially : déclaration d'un ensemble de contraintes qui seront validées une à une. La validation est interrompue dès qu'une contrainte n'est pas respectée.

```
1 #[Assert\Sequentially([
2     new Assert\NotNull,
3     new Assert\Type('string'),
4     new Assert\Length(min: 6),
5 ])]
```

All : cette déclaration permet à ce qu'une contrainte ou à ce qu'un ensemble de contraintes soit appliqué sur un tableau.

```
1 #[Assert\All([
2     new Assert\NotBlank,
3     new Assert\Length(min:6),
4 ])]
5 protected $tableau = [];
```

Count : valide que le nombre d'éléments d'un tableau se situe entre la valeur maximale et minimale.

```
1 #[Assert\Count( min:2, max:10)]
```

UniqueEntity : valide qu'un ou plusieurs champs d'une entité soient uniques. Cette contrainte se situe au-dessus de la classe avec pour paramètre le nom de la propriété.

```
1 #[UniqueEntity('email')]
```

Exemple

Avec toutes les contraintes décrites ci-dessus :

```
1 <?php
2
3 use Symfony\Component\HttpFoundation\Request;
4 use Symfony\Component\Validator\Constraints as Assert;
5 use Symfony\Component\Form\Extension\Core\Type\FileType;
6 use Symfony\Component\Form\Extension\Core\Type\TextType;
7 use Symfony\Component\Form\Extension\Core\Type\EmailType;
8 use Symfony\Component\Form\Extension\Core\Type\CollectionType;
```

```

9 use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
10 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
11
12 class VotreController extends AbstractController
13 {
14     public function votreAction(Request $request)
15     {
16         $maClasse = new MaClasse();
17
18         $form = $this->createFormBuilder($maClasse)
19             ->add('fichier', FileType::class, [
20                 'label' => 'Choisissez un fichier',
21                 'constraints' => [
22                     new Assert\File([
23                         'maxSize' => '1024k',
24                         'mimeType' => [
25                             'application/pdf',
26                             'application/x-pdf'
27                         ],
28                         'mimeTypeMessage' => 'Vérifiez que le format du fichier soit bien en
PDF'
29                     ]),
30                     new Assert\Image([
31                         // options pour la validation d'images
32                         'minWidth' => 200,
33                         'maxWidth' => 400,
34                         'minHeight' => 200,
35                         'maxHeight' => 400,
36                     ]),
37                 ],
38             ])
39             ->add('bic', TextType::class, [
40                 'label' => 'BIC',
41                 'constraints' => [
42                     new Assert\Bic(),
43                 ],
44             ])
45             ->add('champ', TextType::class, [
46                 'label' => 'Champ',
47                 'constraints' => [
48                     new Assert\AtLeastOneOf([
49                         new Assert\Regex([
50                             'pattern' => '/#/',
51                             'message' => 'Le champ doit contenir le caractère "#"
52                         ]),
53                         new Assert\Length([
54                             'min' => 10,
55                             'minMessage' => 'Le champ doit contenir au moins {{ limit }}
caractères'
56                         ]),
57                     ]),
58                 ],
59             ])
60             ->add('autreChamp', TextType::class, [
61                 'label' => 'Autre champ',
62                 'constraints' => [
63                     new Assert\Sequentially([
64                         new Assert\NotNull(),

```

```

65         new Assert\Type(['type' => 'string']),
66         new Assert\Length([
67             'min' => 6,
68             'minMessage' => 'Le champ doit contenir au moins {{ limit }}
caractères'
69         ]),
70     ],
71 ],
72 ])
73 ->add('tableau', CollectionType::class, [
74     'label' => 'Tableau',
75     'entry_type' => TextType::class,
76     'allow_add' => true,
77     'allow_delete' => true,
78     'constraints' => [
79         new Assert\All([
80             new Assert\NotBlank(),
81             new Assert\Length([
82                 'min' => 6,
83                 'minMessage' => 'Le champ doit contenir au moins {{ limit }}
caractères'
84             ]),
85         ]),
86     ],
87 ])
88 ->add('autreTableau', CollectionType::class, [
89     'entry_type' => TextType::class,
90     'allow_add' => true,
91     'constraints' => [
92         new Assert\Count([
93             'min' => 2,
94             'max' => 10,
95             'minMessage' => 'Le tableau doit contenir au moins {{ limit }}
éléments',
96             'maxMessage' => 'Le tableau ne peut contenir plus de {{ limit }}
éléments'
97         ])
98     ]
99 ])
100 ->add('email', EmailType::class, [
101     'constraints' => [
102         new UniqueEntity([
103             'fields' => ['email'],
104             'message' => 'L\'email doit être unique'
105         ])
106     ]
107 ])
108 ->getForm();
109
110 if ($request->isMethod('POST')) {
111     $form->handleRequest($request);
112
113     if ($form->isValid()) {
114         // le formulaire est valide, faire quelque chose ici
115     }
116 }
117
118 return $this->render('mon_template.html.twig', [
119     'form' => $form->createView(),

```

```
120     ] );  
121   }  
122 }
```

C. Validation dans un fichier Twig

Bien que Twig ne soit pas un outil de validation de formulaire, il peut être utilisé pour ajouter des contraintes de validation dans les formulaires générés à partir de templates. Pour cela, Twig fournit des filtres et des fonctions qui peuvent être utilisés pour valider les données saisies par l'utilisateur.

Voici quelques exemples de filtres et de fonctions Twig que vous pouvez utiliser pour ajouter des contraintes de validation :

- Le filtre « **length** » qui permet de limiter la longueur d'une chaîne de caractères.
Par exemple, « `{{ username|length <= 10 }}` » permet de vérifier si la longueur du champ « **username** » est inférieure ou égale à 10 caractères.
- La fonction « **range** » permet de vérifier si une valeur est comprise entre deux limites.
Par exemple, « `{{ age|range(18, 99) }}` » permet de vérifier si l'âge saisi est compris entre 18 et 99 ans.
- Le filtre « **number_format** » permet de formater une valeur numérique.
Par exemple, « `{{ price|number_format(2, '.', ',') }}` » permet de formater le prix avec deux décimales, un point comme séparateur de décimales, et une virgule comme séparateur de milliers.
- La fonction « **date** » permet de vérifier si une date est valide et de la formater.
Par exemple, « `{{ date(date_format='Y-m-d') }}` » permet de vérifier si la date saisie est valide et de la formater au format « **année-mois-jour** ».

Il est également possible d'utiliser des extensions Twig pour ajouter des contraintes de validation plus avancées, telles que des contraintes de type, de format ou de contenu.

En résumé, Twig ne fournit pas de fonctionnalités de validation de formulaire intégrées, mais il offre des filtres, des fonctions et des extensions qui peuvent être utilisés pour ajouter des contraintes de validation dans les formulaires générés à partir de templates.

D. Exercice : Quiz

[solution n°2 p.22]

Question 1

La méthode `add()` est issue de :

- ☐ FormType
- ☐ buildFormInterface
- ☐ Form

Question 2

La méthode `isValid()` renvoie un booléen.

- ☐ Vrai
- ☐ Faux

Question 3

Le paramètre « `{{ limit }}` » fait référence :

- ☐ Au nombre minimum
- ☐ Au nombre maximum
- ☐ Cela dépend du message

Question 4

Cette écriture est-elle correcte dans un fichier PHP ?

```
1 $errors = $validator->validate($file, [
2     new Image([
3         'minWidth' => 200,
4         'maxWidth' => 400,
5         'minHeight' => 200,
6         'maxHeight' => 400,
7     ]),
8 ]);
```

- ☐ Vrai
- ☐ Faux

Question 5

La contrainte de validation « *File* » possède une option spéciale qui se nomme `mimeTypesMessage`.

- ☐ Vrai
- ☐ Faux

IV. Essentiel

Avoir un formulaire c'est bien, mais il faut aussi veiller à ce qui peut y être saisi. La fameuse devise « *never trust user input* », qui signifie ne jamais faire confiance à un utilisateur, ne doit jamais être oubliée par un développeur. En effet, que ce soit à cause d'une mauvaise utilisation ou par malveillance délibérée, un utilisateur peut entrer dans un formulaire des données non utilisables, ou pire, ouvrir une faille de sécurité.

C'est pour cela que des outils de validation ont été créés. Symfony fournit Validator, qui est simple d'installation et d'utilisation.

Il est important de savoir qu'il existe de nombreuses contraintes de validation disponibles. La plupart de ces contraintes ont été présentées dans ce cours, et peuvent être configurées avec différentes options. Par exemple, l'option « *Message* » est applicable à presque toutes les contraintes, mais il existe d'autres options spécifiques à chaque contrainte.

Il y a deux façons d'utiliser les contraintes : en les ajoutant directement à l'entité ou en les définissant dans un fichier PHP qui contient le formulaire. Dans tous les cas, il est important de choisir une méthode et de ne pas écrire les contraintes dans les deux endroits pour éviter les conflits ou les erreurs de validation.

La validation vous posera peut-être quelques problèmes au début, surtout quand il faut parfois choisir son système d'écriture et faire des conversions entre Annotations, Attributs ou PHP, mais vous vous y ferez vite et vous pourrez traiter ainsi toutes les données de manière sécurisée.

V. Auto-évaluation

A. Exercice

Vous êtes un développeur junior dans une agence web. On vous confie à vous ainsi qu'à un autre développeur junior, un projet Symfony 5.4 où il est nécessaire de refaire tous les formulaires, en y incluant des contraintes de validation.

Question 1

[solution n°3 p.23]

Pour commencer, vous devez créer dans un fichier PHP un formulaire d'enregistrement dans lequel il y aura 3 champs :

- Email avec une contrainte email et un message.
- Username avec une contrainte de longueur minimum fixée à 3 avec un message.
- Password avec une contrainte qui vérifie que le mot de passe n'est pas compromis et une autre de longueur fixée avec un minimum de 8.

Question 2

[solution n°4 p.23]

Votre collègue, nouveau dans le domaine du développement, a des difficultés pour effectuer certaines manipulations. D'ailleurs, il ne parvient pas à convertir deux contraintes de validation d'une entité en contrainte de validation pour un fichier PHP. Aidez-le à convertir ces mêmes contraintes.

Les contraintes :

```

1  /**
2   * @Assert\NotNull
3   * @Assert\Type(type="bool")
4   */
5  private $flag;
6
7  /**
8   * @Assert\NotEqualTo(value=0)
9   * @Assert\LessThan(value=10000)
10  */
11 private $number;
```

B. Quiz

Exercice 1 : Quiz

[solution n°5 p.24]

Question 1

Quelle est la bonne conversion de l'attribut ci-dessous en contraintes PHP ?

```

1  ([Assert\All((
2   #[Assert\NotBlank]
3   #[Assert\Length(min: 2, max: 50, minMessage: 'Vous devez avoir plus de {{ limit }}
4  caractères')])
5  private string $name;
```

- ☐ 1. \$builder->add('name', TextType::class, [
 new 'All constraints' => [
 new NotBlank(),
 new Length([
 'min' => 2,
 'max' => 50,
 'minMessage' => 'Vous devez avoir plus de {{ limit }} caractères',
]),
]),

- ```
],
]);
```
- ☐ 2. \$builder->add('name', TextType::class, [
- ```
    'constraints' => [
        new NotBlank(),
        new Length([
            'min' => 2,
            'max' => 50,
            'minMessage' => 'Vous devez avoir plus de {{ limit }} caractères',
        ]),
    ],
]);
```
- ☐ 3. \$builder->add('name', TextType::class, [
- ```
 'constraints' => [
 new All([
 new NotBlank(),
 new Length([
 'min' => 2,
 'max' => 50,
 'minMessage' => 'Vous devez avoir plus de {{ limit }} caractères',
]),
],
],
]);
```

## Question 2

Quelle conversion de cette annotation de contraintes en contraintes de validation dans un contrôleur est exacte ?

```
1 $builder->add('email', EmailType::class, [
2 'constraints' => [
3 new Email(),
4],
5]);
```

- ☐ 1. \$builder->add('email', EmailType::class, [
- ```
    'constraints' => [
        new Email(),
    ],
]);
```
- ☐ 2. #[Assert\Email]
- ☐ 3. \$builder->add('email', EmailType::class, [
- ```
 'Assert\constraints' => [
 new Email(),
],
]);
```

## Question 3

Il existe un outil qui permet de débogger les contraintes.

- ☐ Vrai
- ☐ Faux

Question 4

Quelle contrainte permet de renvoyer true entre 2 et '2' ?

- ☐ EqualTo()
- ☐ IdenticalTo()
- ☐ StringTo()

Question 5

Quelle contrainte de validation permet de vérifier qu'une valeur existe dans un tableau référent ?

- ☐ #[Assert\Choice[]]
- ☐ #[Assert\Array]
- ☐ #[Assert\Tab]

## Solutions des exercices




**Exercice p. 9 Solution n°1****Question 1**

Il est possible de mettre des contraintes de validation sur un formulaire à partir d'un fichier XML.

☒ Vrai

☐ Faux

 Dans un projet Symfony, on ajoute des contraintes de validation dans un fichier XML, YAML, PHP qui est l'option la plus utilisée ou Twig.


**Question 2**

Quelle proposition est fausse sur une propriété d'entité ?

☐ #[Assert\Email(message: "Veuillez écrire l'email au bon format")]

☐ #[Assert\NotBlank(message: "le champ être rempli")]

☒ Aucune des propositions


 Les réponses 1 et 2 sont correctes. Il est possible d'ajouter l'option message à toutes les contraintes de validation proposées par Symfony.

**Question 3**

La contrainte de validation isTrue est fausse si la réponse est une chaîne de caractère string, par exemple : '1'.

☐ Vrai

☒ Faux


 La contrainte de validation istrue attend une réponse 1, '1' ou true.

**Question 4**

On peut mettre plus de 5 contraintes de validation sur une propriété.

☒ Vrai

☐ Faux

 Le nombre de contraintes n'est pas limité.


**Question 5**

Quelle réponse est exacte ?

☐ /\*\*\*@Assert\Blank

☐ #[Assert\NotBlank]

☒ Les réponses 1 et 2 sont exactes


 Il est possible d'ajouter une contrainte de validation sur une propriété avec une annotation ou un attribut.

## Exercice p. 15 Solution n°2

### Question 1

La méthode add() est issue de :


- ☐ FormType
- ☒ buildFormInterface
- ☐ Form

 C'est buildFormInterface qui est appelé sous la forme \$build et fournit la méthode add() pour ajouter un champ dans un formulaire.

### Question 2

La méthode isValid() renvoie un booléen.


- ☒ Vrai
- ☐ Faux

 Si les contraintes de validation ont été respectées, isValid() renvoie true, sinon false.

### Question 3

Le paramètre « **{{ limit }}** » fait référence :

- ☐ Au nombre minimum
- ☐ Au nombre maximum
- ☒ Cela dépend du message


 « *limit* » décrit une valeur minimale ou maximale. Si elle est en référence dans minMessage, elle désigne la valeur minimum. Si cette référence est dans maxMessage, cette valeur désigne le nombre maximum.

### Question 4

Cette écriture est-elle correcte dans un fichier PHP ?

```
1 $errors = $validator->validate($file, [
2 new Image([
3 'minWidth' => 200,
4 'maxWidth' => 400,
5 'minHeight' => 200,
6 'maxHeight' => 400,
7]),
8]);
```

- ☒ Vrai
- ☐ Faux


 Dans un fichier PHP, les contraintes sont dans un tableau et chaque contrainte commence par le mot « **new** ».

### Question 5

La contrainte de validation « *File* » possède une option spéciale qui se nomme mimeTypeesMessage.

☒ Vrai

☐ Faux

 Ce message s'affiche si les types déclarés dans mimeTypees ne correspondent pas.

#### p. 17 Solution n°3

```

1 public function buildForm(FormBuilderInterface $builder, array $options): void
2 {
3 $builder
4 ->add('email', EmailType::class, [
5 'constraints' => [
6 new Email([
7 'message' => "cet adresse mail n'est pas valide"
8])
9]
10])
11 ->add('username', TextType::class, [
12 'constraints' => [
13 new Length([
14 'min' => 3,
15 'minMessage' => 'Vous devez rentrez au moins {{ limit }} caractères'
16])
17]
18])
19 ->add('password', PasswordType::class, [
20 'constraints' => [
21 new NotCompromisedPassword(),
22 new Length([
23 'min' => 8,
24 'minMessage' => 'Vous devez rentrez au moins {{ limit }} caractères'
25])
26]
27])
28];
29
```

#### p. 17 Solution n°4

```

1 $builder
2 ->add('flag', CheckboxType::class, [
3 'constraints' => [
4 new NotNull(),
5 new Type([
6 'type' => 'bool',
7]),
8],
9])
10 ->add('number', IntegerType::class, [
11 'constraints' => [
12 new NotEqualTo([
13 'value' => 0,
14]),
15 new LessThan([

```

```
16 'value' => 10000,
17]),
18],
19);
```

## Exercice p. 17 Solution n°5

### Question 1


Quelle est la bonne conversion de l'attribut ci-dessous en contraintes PHP ?

```
1 ([Assert\All((
2 #[Assert\NotBlank]
3 #[Assert\Length(min: 2, max: 50, minMessage: 'Vous devez avoir plus de {{ limit }}
 caractères')]
4))
5 private string $name;
```

- ☐ 1. \$builder->add('name', TextType::class, [
 new 'All constraints' => [
 new NotBlank(),
 new Length([
 'min' => 2,
 'max' => 50,
 'minMessage' => 'Vous devez avoir plus de {{ limit }} caractères',
 ]),
 ],
]);
- ☐ 2. \$builder->add('name', TextType::class, [
 'constraints' => [
 new NotBlank(),
 new Length([
 'min' => 2,
 'max' => 50,
 'minMessage' => 'Vous devez avoir plus de {{ limit }} caractères',
 ]),
 ],
]);
- ☒ 3. \$builder->add('name', TextType::class, [
 'constraints' => [
 new All([
 new NotBlank(),
 new Length([
 'min' => 2,
 'max' => 50,
 'minMessage' => 'Vous devez avoir plus de {{ limit }} caractères',
 ]),
 ]),
 ],
]);



```
]);
```

 Réponse 3. La syntaxe n'est pas compliquée, mais doit être précise.

### Question 2

Quelle conversion de cette annotation de contraintes en contraintes de validation dans un contrôleur est exacte ?

```
1 $builder->add('email', EmailType::class, [
2 'constraints' => [
3 new Email(),
4],
5]);
```


- ☐ 1. `$builder->add('email', EmailType::class, [  
 'constraints' => [  
 new Email(),  
 ],  
]);`
- ☒ 2. `#[Assert\Email]`
- ☐ 3. `$builder->add('email', EmailType::class, [  
 'Assert\constraints' => [  
 new Email(),  
 ],  
]);`

 La réponse 2 est un attribut et la 3 n'est pas valable.

### Question 3

Il existe un outil qui permet de déboguer les contraintes.


- ☒ Vrai
- ☐ Faux

 Il se nomme debug/validator, il doit s'installer via le terminal de commandes.

### Question 4

Quelle contrainte permet de renvoyer true entre 2 et '2' ?

- ☒ `EqualTo()`
- ☐ `IdenticalTo()`
- ☐ `StringTo()`

 `IdenticalTo` équivaut à l'opérateur d'égalité stricte `===`, tandis que `StringTo` n'existe pas en tant que contrainte de validation.


### Question 5

Quelle contrainte de validation permet de vérifier qu'une valeur existe dans un tableau référent ?

☒ #[Assert\Choice[]]

☐ #[Assert\Array]

☐ #[Assert\Tab]

 Il sera nécessaire de mettre en paramètre soit un tableau défini avant, soit définir le tableau.