Validation des formulaires

Le composant `Validator`

Le composant `Validator` permet de valider des objets `PHP`. Valider les données provenant d'un formulaire est une tâche extrêmement courante pour vérifier les informations entrées par un utilisateur avant par exemple de les enregistrer dans une base de données.

Pour utilise le composant, il suffit d'installer la dépendance :

```
composer req validator
```

Principales contraintes

Le composant `Validator` permet de valider des contraintes.

Une contrainte est simplement une condition, par exemple "est un numéro de carte de crédit valide", ou "est une adresse email valide" etc.

La contrainte `Length`

Prenons un exemple avec la contrainte `Length` qui permet de spécifier qu'une chaîne de caractères doit avoir une longueur minimale et / ou maximale :

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
use Symfony\Component\Form\Extension\Core\Type\CountryType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Validator\Constraints as Assert;;

class DefaultController extends AbstractController
{
  function __construct()
```

```
  {
  }

  #[Route('/', name: 'index')]
  public function index(Request $request)
  {
    $todo = new Todo();
    $form = $this->createFormBuilder($todo)
      ->add('content', TextType::class, [
        'label' => 'Un super label',
        'attr' => [
          'placeholder' => 'Contenu de la todo'
        ],
        'help' => 'Indiquez ce que vous avez à faire'
      ])
      ->add('done', CheckboxType::class, ['required' => false])
      ->add('country', CountryType::class)
      ->add('submit', SubmitType::class)
      ->getForm();

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
      dd($todo);
    }

    return $this->render('page1.html.twig', [
      'myform' => $form->createView()
    ]);
  }
}

class Todo
{
  function  __construct(
    #[Assert\Length(
      min: 10,
      max: 50,
      minMessage: 'Le contenu doit faire au moins {{ limit }} caractè
res',
      maxMessage: 'Le contenu doit faire au plus {{ limit }} caractèr
es',
    )]
    public string $content = '',
    public ?string $country = null,
```

```
        public bool $done = false
    ) {
    }
}
```

Par convention, `Constraints` est renommé en `Assert` simplement pour faciliter la lecture de la contrainte : `use Symfony\Component\Validator\Constraints as Assert`.

Ainsi, notre contrainte `#[Assert\Length` se lit "Impose que la longueur" etc.

`Assert\Length()` **permet de créer une contrainte sur la longueur d'une chaîne de caractères**.

Elle a plusieurs options, mais les principales sont :

- L'option `min` permet de spécifier la taille minimale de la chaîne de caractères. **A noter que si le champ est vide, la contrainte n'est pas prise en considération** (il faut ajouter la contrainte `NotBlank` en plus si vous ne vérifiez pas que le champ est `required`).

- L'option `max` permet de spécifier la taille maximale de la chaîne de caractères. Mêmes considérations si le champ est vide.

- L'option `minMessage` permet de spécifier le message d'erreur affiché lorsque la contrainte de longueur minimale n'est pas respectée. Deux variables sont accessibles `limit` (nombre de caractères minimal spécifié) et `value` (nombre de caractères entrés par l'utilisateur).

- L'option `maxMessage` permet de spécifier le message d'erreur affiché lorsque la contrainte de longueur maximale n'est pas respectée. Deux variables sont accessibles `limit` (nombre de caractères maximal spécifié) et `value` (nombre de caractères entrés par l'utilisateur).

La contrainte `NotBlank`

La contrainte `NotBlank` permet de s'assurer qu'une valeur n'est pas nulle ou vide. Pour une chaîne de caractères, cela signifie qu'elle n'est pas vide ou nulle. Pour tableau, cela signifie qu'il ne soit pas vide. Pour un booléen, que cela ne soit pas `false`.

La seule option que vous devez retenir est `message` qui permet de changer le message d'erreur.

Par exemple, nous pourrions l'ajouter :

```php
<?php

// ...

class Todo
{
  function __construct(
    #[Assert\NotBlank(message: 'Le contenu ne doit pas être vide')]
    #[Assert\Length(
      min: 10,
      max: 50,
      minMessage: 'Le contenu doit faire au moins {{ limit }} caractè
res',
      maxMessage: 'Le contenu doit faire au plus {{ limit }} caractèr
es',
    )]
    public string $content = '',
    public ?string $country = null,
    public bool $done = false
  ) {
  }
}
```

Même si dans ce cas ce n'est pas utile car le champ est déjà requis. Pour vérifier que cela fonctionne, passez le champ à `'required' => false`.

La contrainte `Email`

La contrainte `Email` permet de s'assurer qu'une chaîne de caractères est une adresse email valide.

Les deux options à retenir sont :

- `message` : pour modifier le message d'erreur.

- `mode` : pour modifier le pattern (l'expression régulière) utilisée pour vérifier l'adresse email. La valeur par défaut est `loose` qui vérifie simplement qu'il y a un `@` et un `.`. La valeur `strict` permet de respecter la norme `RFC` pour les adresses emails. La valeur `html5` permet d'utiliser l'expression régulière utilisée par le champ `HTML input` lorsque vous précisez le type `email`.

Voici un exemple :

```php
<?php

use Symfony\Component\Validator\Constraints as Assert;

class User
{
  #[Assert\Email(
    message: '{{ value }} n\'est pas une adresse email valide.',
  )]
  string $email;
}
```

La contrainte `Range`

La contrainte `Range` permet de valider qu'un nombre ou une date au format `DateTime` est dans un intervalle de valeur.

Les options à retenir sont :

- `min` : nombre ou date minimal de l'intervalle.

- `max` : nombre ou date maximum de l'intervalle.

- `minMessage` : message si la valeur entrée est inférieure à la valeur minimale.

- `maxMessage` : message si la valeur entrée est supérieure à la valeur maximale.

- `notInRangeMessage` : message si la valeur entrée n'est pas dans l'intervalle (supérieure ou inférieure).

- `invalidMessage` : message si la valeur entrée n'est pas un nombre et que les limites sont numériques.

- `invalidDateTimeMessage` : message si la valeur entrée n'est pas une `DateTime` et que les limites sont des dates.

Voici un exemple pour un intervalle de nombres :

```php
<?php

use Symfony\Component\Validator\Constraints as Assert;

class User
{
  #[Assert\Range(
```

```
    min: 18,
    max: 120,
    notInRangeMessage: 'L\'âge est incorrect.',
  )]
  protected $age;
}
```

Voici un exemple pour un intervalle de dates :

```
<?php

use Symfony\Component\Validator\Constraints as Assert;

class Todo
{
  #[Assert\Range(
    min: '+1 day',
    max: '+1 year',
    notInRangeMessage: 'La date limite est invalide.',
  )]
  protected $dueDate;
}
```

Nous verrons bien sûr d'autres contraintes selon nos besoins.

Définir les contraintes lors de la construction du formulaire

Au lieu d'utiliser les annotations sur les classes de nos entités, il est également possible de définir les contraintes **directement lors de la création du formulaire, sur les champs**.

En reprenant notre exemple :

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
use Symfony\Component\Form\Extension\Core\Type\CountryType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\HttpFoundation\Request;
```

```php
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;;

class DefaultController extends AbstractController
{
  function __construct()
  {
  }

  #[Route('/', name: 'index')]
  public function index(Request $request)
  {
    $todo = new Todo();
    $form = $this->createFormBuilder($todo)
      ->add('content', TextType::class, [
        'label' => 'Un super label',
        'attr' => [
          'placeholder' => 'Contenu de la todo'
        ],
        'help' => 'Indiquez ce que vous avez à faire',
        'constraints' => [
          new NotBlank(message: 'Le contenu ne doit pas être vide'),
          new Length([
            'min' => 10,
            'max' => 50,
            'minMessage' => 'Le contenu doit faire au moins {{ limit
}} caractères',
            'maxMessage' => 'Le contenu doit faire au plus {{ limit
}} caractères',
          ])
        ],
      ])
      ->add('done', CheckboxType::class, ['required' => false])
      ->add('country', CountryType::class)
      ->add('submit', SubmitType::class)
      ->getForm();

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
      dd($todo);
    }

    return $this->render('page1.html.twig', [
```

```
      'myform' => $form->createView()
    ]);
  }
}

class Todo
{
  function  __construct(
    public string $content = '',
    public ?string $country = null,
    public bool $done = false
  ) {
  }
}
```

L'utilisation des annotations est obligatoires pour valider des objets qui ne sont pas des formulaires (par exemple une requête d'`API`).

Dans le cadre des formulaires, la validation peut se faire sur les entités ou sur les champs. Il n'y a pas de meilleure pratique.

Utiliser le service `ValidatorInterface`

Il est possible d'utiliser le service `ValidatorInterface` afin de gérer soit même les erreurs sans les renvoyer à `Twig` pour l'affichage.

C'est particulièrement utile si vous n'êtes pas dans le cadre d'un formulaire mais que vous devez valider un objet, par exemple lors d'une requête d'`API`.

```
<?php

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Validator\Validator\ValidatorInterface;

public function todo(ValidatorInterface $validator)
{
  $todo = new Todo();

  $errors = $validator->validate($todo);

  if (count($errors) > 0) {

    dd($errors);
```

```
            return new Response('Une erreur est survenue'));
        }
    }
```

La méthode `$validator->validate()` retourne un tableau contenant des violations de contraintes.

Nous ne détaillerons pas plus maintenant car nous sommes dans le chapitre relatif aux formulaires.