



Simon GAUTIER



Gestion des formulaires

Paramétrage

Construction d'un formulaire dans un contrôleur / dans une classe dédiée

Validation d'un objet avec des contraintes

Traitement d'un formulaire

Affichage d'un formulaire

Paramétrage de Symfony

- Installer la dépendance nécessaire au fonctionnement des formulaires :

```
$ composer req form
```

Le composant Form

Principe de fonctionnement

- Philosophie de Symfony : un formulaire se construit sur un objet existant, et l'objectif est d'hydrater cet objet
 - Hydrater l'objet = alimenter ses propriétés (souvent via les setters)
 - Hydratation de l'objet possible à partir des données d'une requête HTTP ou directement dans le code
- Objet existant :
 - Une entité est un objet qu'il est intéressant d'hydrater (ex : formulaire d'ajout d'un véhicule)
 - Toutefois, il n'y a aucune obligation sur la nature de l'objet. D'ailleurs, le composant Form de Symfony est totalement indépendant de Doctrine
- Une fois l'objet hydraté par le formulaire, on en fait « ce qu'on veut » :
 - Sauvegarde en base, envoi d'un mail, ...

Exemple d'objet manipulable par le composant Form

```
namespace App\Entity;

class Tire
{
    // La propriété est publique ==> le composant Form peut l'exploiter
    public $brandName;

    // La propriété est privée mais il y a un getter / setter ==> le composant Form peut l'exploiter
    private $price;

    /**
     * @return mixed
     */
    public function getPrice()
    {
        return $this->price;
    }

    /**
     * @param mixed $price
     * @return Tire
     */
    public function setPrice($price)
    {
        $this->price = $price;
        return $this;
    }
}
```

Construction d'un formulaire dans un contrôleur

Recommandé uniquement pour les formulaires très simples

```
class TireController extends Controller
{
    /**
     * @Route("/tire/new", name="tire_new")
     * @return \Symfony\Component\HttpFoundation\Response
     */
    public function add()
    {
        $tire = new Tire(); // Initialisation de l'objet sur lequel le formulaire va reposer
        $tire->brandName = 'Michelin'; // Cette initialisation donnera une valeur par défaut au champ (facultatif)
        $tire->setPrice(16); // Cette initialisation donnera une valeur par défaut au champ (facultatif)

        $form = $this->createFormBuilder($tire) // Ajout de chacun des champs du formulaire via la méthode add()
            ->add('brandName', TextType::class)
            ->add('price', MoneyType::class, ['currency' => 'EUR']) // Certains champs peuvent avoir des paramètres
            ->add('send', SubmitType::class, ['label' => 'Add a new tire'])
            ->getForm(); // Création du formulaire

        return $this->render('tire/new.html.twig', array(
            'form' => $form->createView() // Création d'une vue pour le formulaire ==> nous verrons plus tard comment afficher
                                          // le formulaire dans Twig
        ));
    }
}
```

Construction d'un formulaire dans un contrôleur

Inconvénients

- Inconvénient FormBuilder : le formulaire est décrit entièrement dans l'action du contrôleur
 - ➔ Si on veut le réutiliser dans un autre contrôleur, le code est dupliqué
- Solution : externaliser la définition du formulaire dans une classe dédiée, nommée par convention « **MonFormulaireType** » (« **Type** » toujours en suffixe) et placée par convention dans le répertoire « **src/Form** »
- Générer cette classe dédiée : via la console !

```
php bin/console make:form
```

 - ➔ Donner le nom de la classe
- Puis :
 - Lister les champs du formulaire dans **buildForm()** comme dans le contrôleur
 - Méthode **configureOption()** : permet de définir l'objet autour duquel le formulaire est construit (facultatif)

Construction d'un formulaire dans une classe dédiée

Equivalent au précédent exemple

```
namespace App\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\MoneyType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;

class TireType extends AbstractType // Hériter de la classe \Symfony\Component\Form\AbstractType et implémenter buildForm()
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('brandName', TextType::class) // Ajout des champs de la même manière que dans le contrôleur
            ->add('price', MoneyType::class, ['currency' => 'EUR']);
        // Pas besoin d'appeler getForm() dans ce cas
    }
}

public function addBis()
{
    $tire = new Tire(); // Aucun changement
    $tire->brandName = 'Michelin';
    $tire->setPrice(16);

    $form = $this->createForm(TireType::class, $tire);
    // Eventuellement, ajout de champs supplémentaires (classiquement les boutons de validation => ne pas polluer le formulaire)
    // Il est également possible de supprimer des champs
    $form->add('send', SubmitType::class, ['label' => 'Add a new tire']);

    return $this->render('tire/new.html.twig', array('form' => $form->createView())); // Aucun changement
}
```

Construction d'un formulaire

Quels types disponibles ?

- <https://symfony.com/doc/current/reference/forms/types.html>
 - Pas possible de décrire en détail chaque champ dans ce cours, la doc le fait mieux 😊
- Champs texte :
 - **TextType, TextareaType, EmailType, IntegerType, MoneyType, NumberType, PasswordType, PercentType, SearchType, UrlType, RangeType**
- Champs de choix :
 - **ChoiceType, EntityType, CountryType, LanguageType, LocaleType, TimezoneType, CurrencyType**
- Champs de date / heure :
 - **DateType, DateIntervalType, DateTimeType, TimeType, BirthdayType**
- Autres champs : **CheckboxType, FileType, RadioType**
- Champs multiples : **CollectionType, RepeatedType**
- Champs cachés : **HiddenType**
- Boutons : **ButtonType, ResetType, SubmitType**
- Champs de base : **FormType**

Construction d'un formulaire

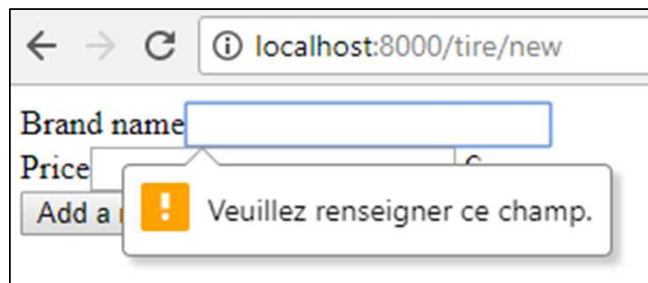
- Le formulaire créé de cette manière n'est pas « fonctionnel » (il ne se passe rien si on le valide)
- Les valeurs par défaut du formulaire sont les valeurs des propriétés de l'objet rattaché au formulaire
 - Soit initialisées par le développeur (appel aux setters)
 - Soit initialisées au niveau de l'objet lui-même (ex : constructeur)
- Retenir que la création d'un champ se fait via la méthode `add()` sur le FormBuilder. 3 paramètres sont disponibles :
 - Nom du champ
 - Type du champ. Il ne s'agit pas du type HTML, mais du type sémantique. Exemple : `DateTime` génère 3 champs de type `select` ! C'est au développeur de choisir le bon type en fonction de la propriété de l'objet qu'il faut hydrater
 - Options du champ (sous forme de tableau). Les options sont liées au type du champ qu'on manipule

Validation d'un objet

- Pourquoi valider les données ?
 - Never trust user input ! (erreurs possibles / utilisateurs malintentionnés qui cherchent des failles)
- Validation dans Symfony : service « **validator** »
 - Service indépendant des formulaires : on peut valider un objet sans que celui-ci ait été hydraté par un formulaire
- Valider un objet = vérifier un ensemble de règles :
 - Longueur d'une chaîne de caractères, email correct, ...
- Concrètement : on définit les règles au niveau d'un objet (ex : une entité) puis on déclenche la validation qui contrôle ces règles
 - Recommandation : utiliser les annotations
 - Il est également possible de passer par du YAML ou XML ➔ cf. doc

Validation d'un objet

- Important ! Même si les formulaires sont générés en HTML5 et que des contrôles sont faits automatiquement côté client, cela ne dispense pas de faire des contrôles côté serveur
 - C'est même **obligatoire car la requête HTTP peut être trafiquée** par l'utilisateur. Rappel : Never trust user input !



Les contrôles côté client ne sont qu'un élément de confort pour l'utilisateur mais en aucun cas un gage de sécurité pour notre application.

Validation d'un objet

\$ composer req validator

- Important : pour pouvoir utiliser les annotations :

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

- Syntaxe de l'annotation (en fonction des options) :

```
@Assert\NomContrainte()
```

```
@Assert\NomContrainte(valeur option par défaut)
```

```
@Assert\NomContrainte(option1="valeur1", option2="valeur2", ...)
```

- Une option disponible pour beaucoup de contraintes : « **message** »

- Permet de préciser le message d'erreur à restituer (parfois avec une variable). Exemple :

```
@Assert\Email(message="L'email {{ value }} n'est pas correct")
```

Validation d'un objet

Exemple

```
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Tire
{
    /**
     * @Assert\NotBlank(message="Le nom de la marque est obligatoire")
     * @Assert\Length(max="64")
     */
    public $brandName;

    /**
     * @Assert\NotBlank()
     * @Assert\Range(
     *     min="1",
     *     max="999",
     *     minMessage="Le prix ne peut pas être inférieur à 1€",
     *     maxMessage="Le prix ne peut pas être supérieur à 1000€"
     * )
     */
    private $price;

    // ...
}
```

Un champ peut ne pas avoir de contrainte de validation

Un champ peut cumuler les contraintes de validation

Remarque sur les traductions : utiliser des codes et pas des textes complets :

message="Le nom de la marque est obligatoire"

À remplacer par :

message="tire.name.not_blank"

➔ cf. chapitre i18n

Validation d'un objet

Contraintes disponibles

- Toutes les contraintes disponibles et leurs options en détails : cf. documentation :
 - <https://symfony.com/doc/current/reference/constraints.html>
 - Dans ce cours, on ne présente que quelques contraintes et leurs principales options
- Contraintes de base :
 - **NotBlank** : ni une chaîne vide, ni false, ni null
 - **Blank** : chaîne vide ou false ou null
 - **NotNull** : valeur strictement différente de null
 - **IsNull** : valeur strictement égale à null
 - **IsTrue** : booléen true ou entier 1 ou chaîne « 1 »
 - **IsFalse** : booléen false ou entier 0 ou chaîne « 0 »
 - **Type** : Vérifie que la valeur est du type donné
 - Option : le type attendu (array, bool, numeric, ... ➔ voir tous les types disponibles : <https://symfony.com/doc/current/reference/constraints/Type.html>)

Validation d'un objet

Contraintes disponibles

- Contraintes sur les chaînes de caractères :
 - **Email** : email valide
 - Options : **checkMX** (**false**), **strict** (**false**), **checkHost** (**false**)
 - **Length** : valider la longueur d'une chaîne
 - Options : **min**, **max**, **minMessage**, **maxMessage**, **exactMessage**, **charset** (**UTF-8**) pour préciser le charset de la chaîne dont il faut compter les caractères
 - ➔ Les messages peuvent utiliser la variable « **{{ limit }}** »
 - **Url** : valider une URL
 - Options : **protocols** (**array('http', 'https')**), **checkDNS** (**false**)
 - **Regex** : valider une chaîne à partir d'une expression régulière
 - Options : **pattern** pour l'expression à valider, **match** (**true**) pour préciser si on veut que la valeur corresponde ou non à l'expression régulière
 - **Ip** : valide une adresse IP
 - Option : **version** (« **4** » par défaut, il y a beaucoup d'autres options disponibles, cf. documentation)
 - **Uuid** : vérifie que la chaîne est un UUID (cf. documentation)

Validation d'un objet

Contraintes disponibles

- Contraintes sur les nombre :
 - **Range** : vérifie qu'un nombre ou une date est dans un intervalle donné
 - Options : **min**, **max**, **minMessage**, **maxMessage**, **invalidMessage** (si la valeur n'est pas un nombre)
 - ➔ Les messages **minMessage** et **maxMessage** peuvent utiliser la variable « **{{ limit }}** »
- Contraintes sur les dates :
 - **Date** : vérifie que la valeur est un objet de type DateTime ou une chaîne de caractères du type « YYYY-MM-DD »
 - **DateTime** : vérifie que la valeur est un objet de type DateTime ou une chaîne de caractères du type « YYYY-MM-DD HH:MM:SS »
 - **Time** : vérifie que la valeur est un objet de type DateTime ou une chaîne de caractères du type « HH:MM:SS »

Validation d'un objet

Contraintes disponibles

- Contraintes par comparaison :
 - **EqualTo / NotEqualTo** : comparaison non stricte
 - ➔ utilise « == » et « != »)
 - Option : **value**, utilisation de la variable `{{ compared_value }}` pour le message
 - **IdenticalTo / NotIdenticalTo** : comparaison stricte
 - ➔ utilise « === » et « !== »)
 - Option : **value**, utilisation des variables `{{ compared_value_type }}` et `{{ compared_value }}` pour le message
 - **LessThan / GreaterThan** : strictement plus petit / plus grand que
 - Option : **value**, utilisation de la variable `{{ compared_value }}` pour le message
 - **LessThanOrEqual / GreaterThanOrEqual** : inférieur / supérieur ou égal à
 - Option : **value**, utilisation de la variable `{{ compared_value }}` pour le message

Validation d'un objet

Contraintes disponibles

- Contraintes sur les fichiers :
 - **File** : vérifie un fichier (la valeur comparée peut être une chaîne de caractères représentant un chemin vers un fichier ou objet Symfony File)
 - Beaucoup d'options (cf. documentation) : **maxSize**, **binaryFormat**, **mimeTypes**, **maxSizeMessage**, **mimeTypesMessage**, **disallowEmptyMessage**, **notFoundMessage**, **notReadableMessage**, **uploadIniSizeErrorMessage**, **uploadFormSizeErrorMessage**, **uploadErrorMessage**
 - **Image** : vérifie une image (idem File) mais avec les options **mimeTypes** et **mimeTypesMessage** initialisées pour les images
 - Autres options disponibles pour les images : **minWidth**, **maxWidth**, **maxHeight**, **minHeight**, **maxRatio**, **minRatio**, **allowSquare**, **allowLandscape**, **allowPortrait**, **detectCorrupted**, **sizeNotDetectedMessage**, **maxWidthMessage**, **minWidthMessage**, **maxHeightMessage**, **minHeightMessage**, **maxRatioMessage**, **minRatioMessage**, **allowSquareMessage**, **allowLandscapeMessage**, **allowPortraitMessage**, **corruptedMessage**

Validation d'un objet

Contraintes disponibles

- **Contraintes sur des collections :**
 - **Choice** : la valeur doit faire partie d'une liste donnée
 - **Collection** : vérifier indépendamment chaque membre d'une collection de valeurs avec des contraintes natives
 - **Count** : vérifier le nombre d'éléments de la collection donnée
 - **UniqueEntity** : vérifier qu'un N-uplet d'attributs de l'entité est unique
 - **Language** : vérifier que la valeur est un code de langue valide (Unicode language identifier)
 - **Locale** : vérifier que la valeur est une locale valide :
 - Soit 2 lettres (ex : **fr**) → norme ISO 639-1
 - Soit le format avec underscore (ex : **fr_FR**) → norme ISO 3166-1 alpha-2
 - **Country** : vérifier que la valeur est un pays valide (la valeur respecte la norme ISO 3166-1 alpha-2)

Validation d'un objet

Contraintes disponibles

- Contraintes sur valeurs monétaires :
 - **Bic, CardScheme, Currency, Luhn, Iban, Isbn, Issn**
- Autres contraintes :
 - **Callback** : vu plus loin dans le cours 😊
 - **Expression** : validation conditionnelle entre attributs de l'objet
 - Exemple : si attribut \$att1 vaut X, alors vérifier que \$att2 vaut Y
 - **All** : sur un tableau, appliquer les mêmes validations sur tous les éléments
 - **UserPassword** : vérifier que la valeur correspond au mot de passe courant de l'utilisateur ➔ classiquement utilisé dans les formulaires où l'utilisateur doit donner son mot de passe pour pouvoir valider le formulaire
 - **Valid** : vu juste après 😊

Validation d'un objet

Zoom sur la contrainte **Valid**

- Problématique : si un attribut d'un objet A est un autre objet B qui a lui-même ses règles de validation
 - La contrainte **Valid** permet de déclencher la validation des règles de B depuis A
 - Exemple sur l'objet Vehicle :

```
class Vehicle
{
    // ...
    /**
     * @var VehicleModel
     * @ORM\ManyToOne(targetEntity="App\Entity\VehicleModel", inversedBy="vehicles")
     * @Assert\Valid()
     */
    private $model;

    /**
     * @var PersistentCollection
     * @ORM\ManyToMany(targetEntity="App\Entity\VehicleEquipment")
     * @ORM\JoinTable(name="asso_vehicle_equipment")
     * @Assert\Valid()
     */
    private $equipments;
    // ...
}
```

Demander la validation
des objets liés

```
class VehicleModel
{
    // ...
    /**
     * @var string
     * @ORM\Column()
     * @Assert\NotBlank()
     */
    private $name;
    // ...
}
```

Contrainte sur le
modèle de véhicule

```
class VehicleEquipment
{
    // ...
    /**
     * @var string
     * @ORM\Column()
     * @Assert\NotBlank()
     */
    private $name;
    // ...
}
```

Contrainte sur
l'équipement

Validation d'un objet

Zoom sur la contrainte **Callback**

- Permet de définir le fonctionnel souhaité pour une contrainte
 - Permet de définir des contraintes qui n'existent pas nativement
 - Permet de comparer les attributs entre eux dans le callback
 - Avantage : on peut rattacher l'erreur à un attribut en particulier
 - Exemple dans VehicleEquipment : interdire une description identique au name :

```
class VehicleEquipment
{
    // ...

    /**
     * @Assert\Callback()
     */
    public function isDescriptionValid(ExecutionContextInterface $context)
    {
        if($this->name == $this->description) {
            // La règle est violée ==> définition de l'erreur
            $context
                ->buildViolation("Il est interdit d'utiliser la même valeur pour le nom et la description") // Message
                ->atPath('description') // Préciser l'attribut de l'objet qui est violé
                ->addViolation()
            ;
        }
    }
}
```

Pour aller plus loin, on pourrait créer un service afin d'accéder à plus de choses que l'objet lui-même → cf. chapitre « Les formulaires (avancé) »

Validation d'un objet

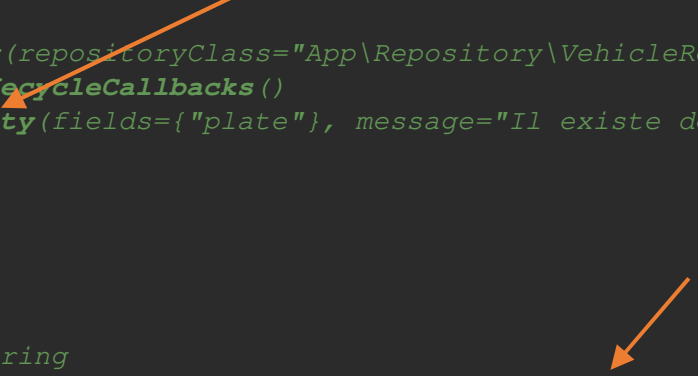
Zoom sur la contrainte **UniqueEntity**

- Vérifier qu'un attribut (ou N-uplet d'attributs) est unique parmi toutes les entités existantes
 - Fait partie du bridge entre Doctrine et Symfony (n'est pas lié au composant Validator)
 - Déclaration particulière : annotation **@UniqueEntity** au niveau de la classe
 - A ajouter : `use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;`
 - Exemple avec Vehicle : interdire 2 plaques d'immatriculation identiques :

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\VehicleRepository")
 * @ORM\HasLifecycleCallbacks()
 * @UniqueEntity(fields={"plate"}, message="Il existe déjà un véhicule avec cette plaque d'immatriculation")
 */
class Vehicle
{
    // ...

    /**
     * @var string
     * @ORM\Column(type="string", length=10, unique=true)
     */
    private $plate;

    // ...
}
```



Pour être cohérent, imposer également cette logique à Doctrine

Validation d'un objet

Méthodes automatiques

- Dans un objet qui doit être validé, créer une méthode **getXxx()** ou **isXxx()** ou **hasXxx()** avec une annotation **@Assert**

- Cette méthode sera alors automatiquement déclenchée à la validation de l'objet

- Exemple :

```
/**
 * @Assert\IsTrue(message="Le prix n'est pas correct")
 */
public function isPrice()
{
    return is_double($this->price);
}
```


- Quid des messages d'erreur :

- Si « Xxx » correspond à un attribut de l'objet (ex : **isPrice()**) → message lié à l'attribut concerné (l'erreur serait affichée au niveau du champ de formulaire correspondant).
- Sinon → message global à l'objet (l'erreur serait affichée en haut du formulaire).

Validation d'un objet

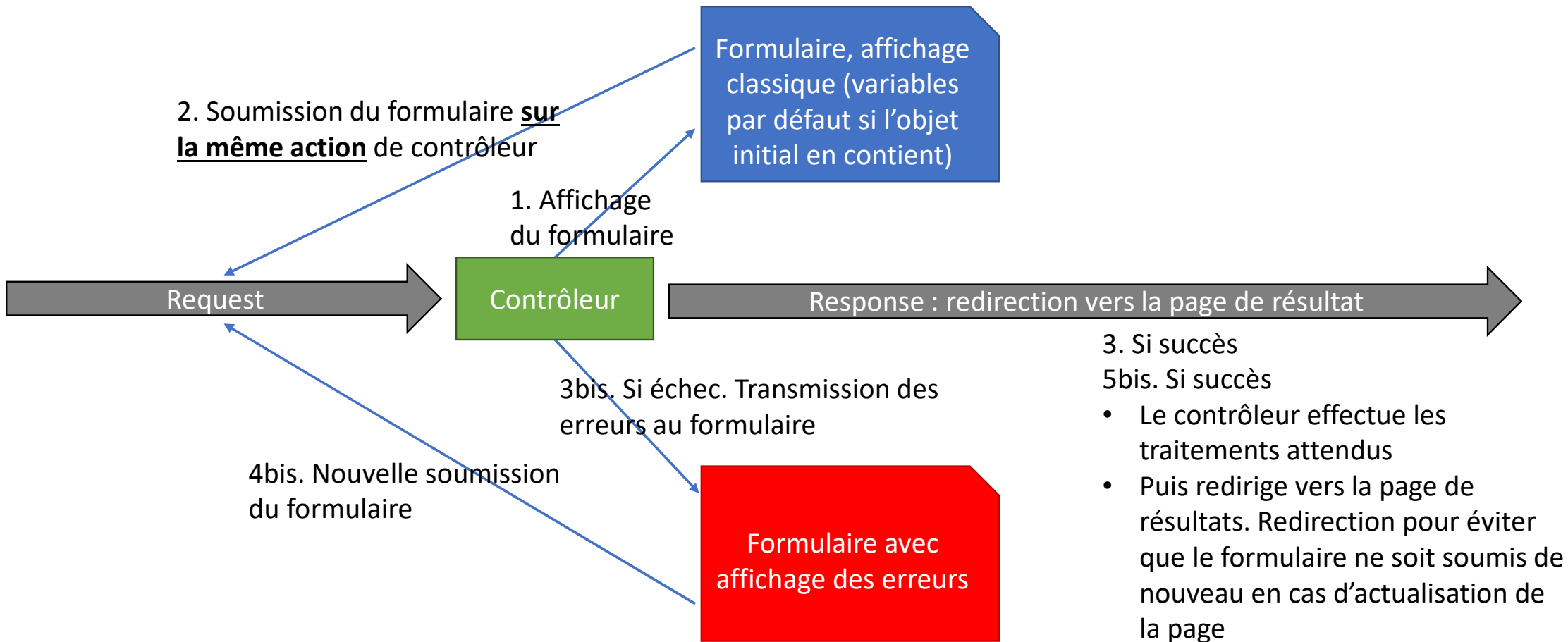
Déclencher la validation

```
public function testValid(ValidatorInterface $validator)
{
    // Le validateur est injecté dans le constructeur
    // Il est également possible d'y accéder via le service : $validator = $this->get('validator');
    $tire = new Tire();
    dump($validator->validate($tire));
    return new Response();
}
```



```
TireController.php on line 101:
ConstraintViolationList {#874 ▼
  -violations: array:2 [▼
    0 => ConstraintViolation {#740 ▼
      -message: "Le nom de la marque est obligatoire"
      -messageTemplate: "Le nom de la marque est obligatoire"
      -parameters: array:1 [▶]
      -plural: null
      -root: Tire {#971 ▶}
      -propertyPath: "brandName"
      -invalidValue: null
      -constraint: NotBlank {#694 ▶}
      -code: "c1051bb4-d103-4f74-8988-acbcafc7fdc3"
      -cause: null
    }
    1 => ConstraintViolation {#881 ▼
      -message: "Cette valeur ne doit pas être vide."
      -messageTemplate: "This value should not be blank."
      -parameters: array:1 [▶]
      -plural: null
      -root: Tire {#971 ▶}
      -propertyPath: "price"
      -invalidValue: null
      -constraint: NotBlank {#846 ▶}
      -code: "c1051bb4-d103-4f74-8988-acbcafc7fdc3"
      -cause: null
    }
  ]
}
```

Traitement d'un formulaire



Traitement d'un formulaire

```
/**
 * @Route("/tire/new", name="tire_new")
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function add(Request $request)
{
    $tire = new Tire();
    $form = $this->createForm(TireType::class, $tire);
    $form->add('send', SubmitType::class, ['label' => 'Add a new tire']);
    $form->handleRequest($request); // Alimentation du formulaire avec la Request

    if ($form->isSubmitted() && $form->isValid()) {
        // Le formulaire vient d'être soumis et il est valide ==> $tire est hydraté avec les données saisies

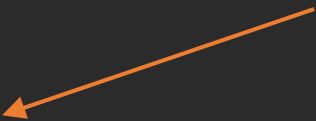
        // Traitement des données du formulaire...

        return $this->redirectToRoute('tire_new_success');
    }

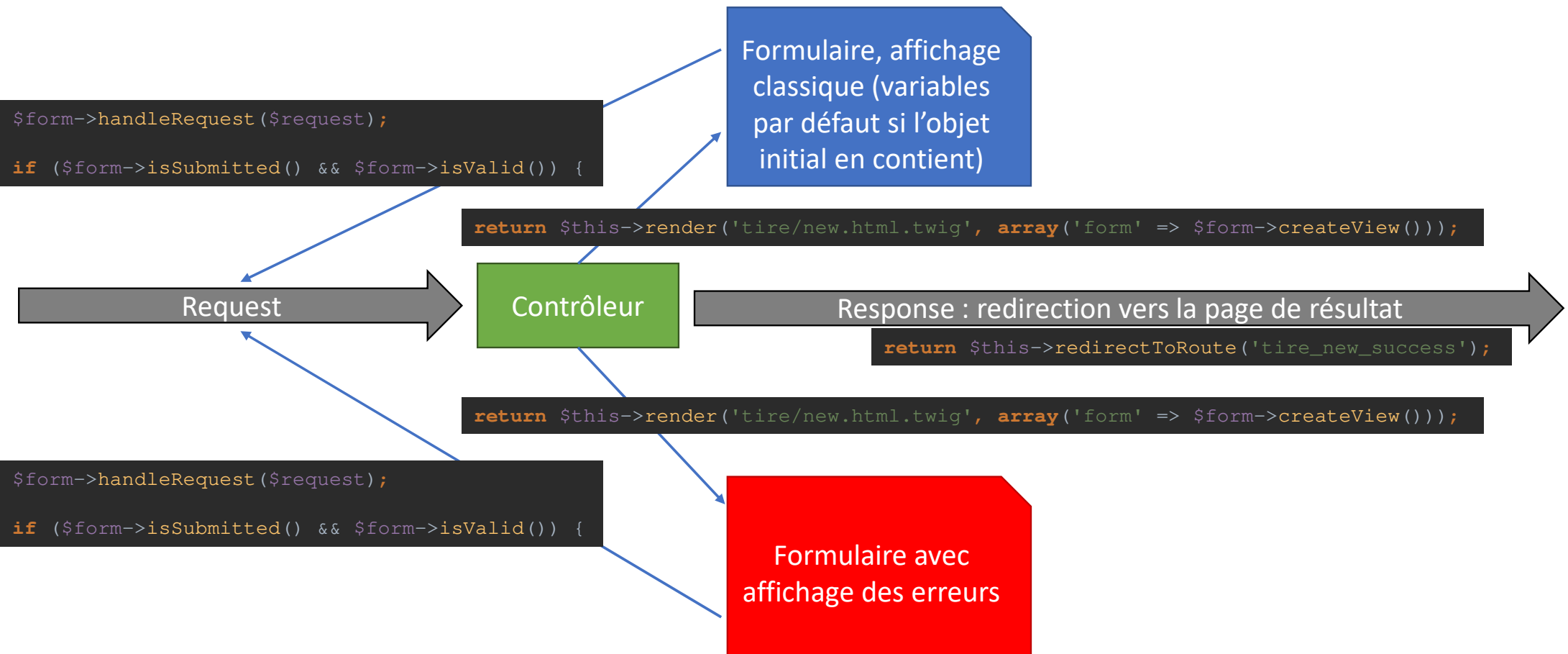
    // Affichage du formulaire initial (requête GET) OU affichage du formulaire avec erreurs après validation (requête POST)
    return $this->render('tire/new.html.twig', array('form' => $form->createView()));
}

/**
 * @Route("/tire/new_success", name="tire_new_success")
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function addSuccess()
{
    return $this->render('tire/success.html.twig');
}
```

Noter que la méthode **handleRequest()** déclenche implicitement la validation de l'objet lié au formulaire (cf. méthode **validate()** présenté précédemment)



Traitement d'un formulaire



Affichage d'un formulaire

- Affichage standard et complet (**form** est le nom du formulaire transmis à Twig) : `{{ form(form) }}`
 - Affiche chacun des champs du formulaire
 - Affiche les labels de chacun des champs
 - Affiche les messages d'erreur de chacun des champs s'il y en a
 - Affiche les messages d'erreur généraux s'il y en a
- Méthode simple mais limitée : on ne peut pas personnaliser soi-même l'agencement des champs ni avoir des particularités d'affichage

Affichage d'un formulaire

```
{{ form_start(form) }}
{# Affiche la balise <form>, définit les attributs action et method + si nécessaire l'attribut enctype #}

{{ form_errors(form) }}
{# Affiche les erreurs globales du formulaire #}

{{ form_row(form.brandName) }}
{# Affiche le champ de manière complète : label + widget (code HTML du champ) + erreurs associées au champ s'il y en a #}

<div>
    {{ form_label(form.price) }}
    {# Affiche uniquement le label du champ #}

    {{ form_errors(form.price) }}
    {# Affiche les erreurs associées au champ s'il y en a #}

    {{ form_widget(form.price) }}
    {# Affiche le widget (code HTML du champ) #}
</div>

{{ form_rest(form) }}
{# Affiche tous les champs non explicitement affichés précédemment #}

{{ form_end(form) }}
{# Affiche la balise fermante </form> et surtout chaque champ qui n'a pas été explicitement affiché (sauf si form_rest()
   a déjà été utilisé) ==> Utile notamment pour le champ CSRF qui n'est jamais explicitement affiché #}
```

Affichage d'un formulaire - Options

- Pour la plupart des fonctions d'affichage, des options sont disponibles. Quelques exemples ci-dessous :
 - Détails : <https://symfony.com/doc/current/forms.html>

```
{{ form(form, {  
    'action': '...',  
    'method': 'GET'  
}) }}
```

```
{{ form_start(form, {  
    'action': '...',  
    'method': 'GET'  
}) }}
```

```
{{ form_label(form.price,  
    'Le prix du pneu',  
    {'label_attr': {  
        'class': 'my_html_class'  
    }}  
) }}
```

```
{{ form_widget(form.price,  
    {'attr': {  
        'class': 'my_html_class'  
    }}  
) }}
```

Affichage d'un formulaire

- Remarque sur le CSRF (Cross Site Request Forgeries) :
 - Permet de vérifier que l'internaute qui valide le formulaire est bien celui qui l'a affiché (sécurité) ➔ géré automatiquement par Symfony
- Création de types de champs personnalisés :
 - Si l'ensemble des types natifs de Symfony ne suffit pas, il est possible de créer ses propres types
 - Principe : création d'une classe pour le type de champ + un template le représentant
 - Pour les détails, cf. documentation :
https://symfony.com/doc/current/form/create_custom_field_type.html

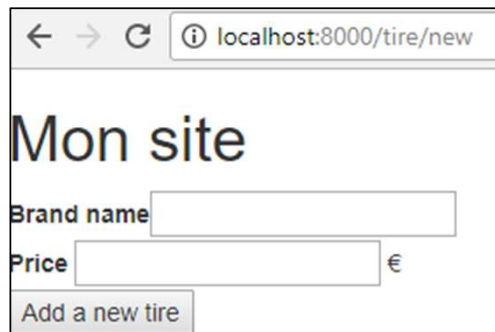
Affichage d'un formulaire

Thème général

- Il est possible de préciser dans la configuration générale le thème utilisé pour nos formulaires
 - Si le site qu'on développe fonctionne avec Bootstrap, il existe des thèmes compatibles
 - Thèmes disponibles : http://symfony.com/doc/current/form/form_customization.html#what-are-form-themes
 - Action à faire dans **config/packages/twig.yaml** dans la section « **form_themes** » : préciser le thème de formulaire à utiliser (un « simple » template Twig qui donne une structure HTML au formulaire) :

```
twig:
    # ...
    form_themes:
        - 'bootstrap_3_layout.html.twig'
```

- Avant / après :



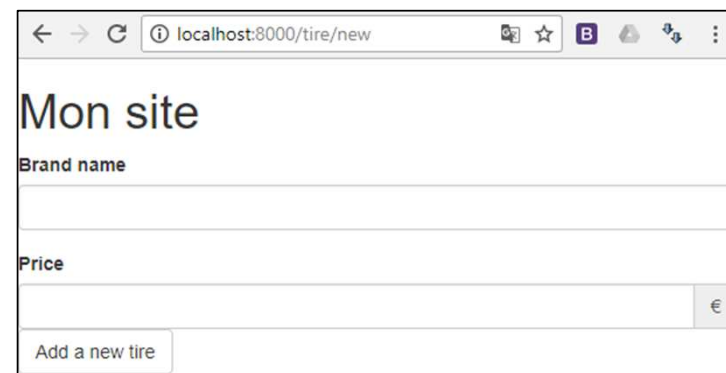
localhost:8000/tire/new

Mon site

Brand name

Price €

Add a new tire



localhost:8000/tire/new

Mon site

Brand name

Price €

Add a new tire

Les formulaires

Le profiler à la rescousse

The screenshot displays the Symfony Profiler interface. At the top, a summary bar shows '200 @ tire_new 364 ms 2.0 MB' and a red arrow points to the 'Forms' icon. Below this, a sidebar on the left contains navigation links: 'Request / Response', 'Performance', 'Validator', 'Forms' (highlighted with a blue arrow), 'Exception', 'Logs' (3 items), 'Events', 'Routing', 'Cache', 'Translation' (3 items), and 'Security'. The main content area is titled 'Forms' and shows a list of forms with 'tire' selected. The details for the 'tire' form are displayed on the right, including its class 'App\Form\TireType' and a table of default data.

Number of forms 1
Number of errors 0

200 @ tire_new 364 ms 2.0 MB 1

Symfony Profiler

http://localhost:8000/tire/new
Method: GET HTTP Status: 200 IP: ::1 Profiled on: Sun, 04 Feb 2018 11:51:26 +0000 Token: 8417d4

Last 10 Latest Search

Request / Response
Performance
Validator
Forms
Exception
Logs 3
Events
Routing
Cache
Translation 3
Security

Forms

tire

- brandName
- price
- send

tire ("App\Form\TireType")

Default Data -

Property	Value
Model Format	same as normalized format
Normalized Format	Tire {#770 ▶}
View Format	same as normalized format

Submitted Data -

This form was not submitted.

Passed Options -

TP formulaires



- Créer une classe **ArticleType** et y lister les champs suivants : **title**, **content**, **author**, **nb_views**, **published** avec les types attendus
- Ajouter les validations suivantes sur l'objet **Article** :
 - Champs obligatoires : **title**, **nb_views**
 - Remarque : pour rendre un champ facultatif, utiliser les options du champ (car par défaut les champs sont obligatoires) :

```
->add('price', MoneyType::class, ['currency' => 'EUR', 'required' => false])
```
 - Le champ **content** ne doit pas avoir la même valeur que le champ **title**
 - Le champ **nb_views** doit être un entier strictement positif
- Dynamiser les actions **addAction()** et **editAction()** du contrôleur des articles du blog :
 - Ne pas oublier le champ de validation du formulaire
 - Pour l'ajout, donner « 1 » comme valeur par défaut à **nb_views**
 - Pour l'édition, charger au préalable l'objet Article avec Doctrine
- Affichage du formulaire : dans le template Twig prévu pour notre formulaire, afficher les champs du formulaire en utilisant dans un premier temps pour tester la fonction **form()** et ensuite en utilisant toutes les méthodes **form_xxx()** présentées durant le cours
 - Utiliser la fonction **form_row()** pour tous les champs sauf pour un où il faut utiliser les sous-fonctions permettant de séparer le label, le champ et les erreurs